Project no.:	**619209**
Project full title:	**Analysis of MassIve Data STreams**
Project Acronym:	**AMIDST**
Deliverable no.:	**D2.3**
Title of the deliverable:	**A software library implementation of the modelling framework**

Contractual Date of Delivery to the CEC:	**31.03.2015**
Actual Date of Delivery to the CEC:	**31.03.2015**
Organisation name of lead contractor for this deliverable:	**AAU**
Author(s):	**Hanen Borchani, Antonio Fernández, Helge Langseth, Anders L. Madsen, Ana M. Martínez, Andrés Masegosa, Thomas D. Nielsen, Antonio Salmerón**
Participants(s):	**P01, P02, P03, P04, P05, P06, P07**
Work package contributing to the deliverable:	**WP2**
Nature:	**R**
Version:	**1.0**
Total number of pages:	**20**
Start date of project:	**1st January 2014 Duration: 36 month**

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

In this document, we describe the software library implementation of the AMIDST modelling framework. We give a general overview of the developed core components consisting of the data structures and the database functionalities related to the AMIDST learning and inference algorithms. In addition, we present the HUGIN AMIDST API ensuring the interaction between the open source ADMIST toolbox and HUGIN software.

Keyword list: AMIDST modelling framework, software library, implementation, core components.

Contents

Document history

Version	Date	Author (Unit)	Description
v0.3	10/03/2015	All consortium members	The software library implementation for the AMIDST modelling framework discussed and established
v0.6	20/03/2015	Hanen Borchani, Antonio Fernández, Helge Langseth, Anders L. Madsen, Ana M. Martínez, Andrés Masegosa, Thomas D. Nielsen, Antonio Salmerón	Initial version of document finished and reviewed
v1.0	31/03/2015	Hanen Borchani, Antonio Fernández, Helge Langseth, Anders L. Madsen, Ana M. Martínez, Andrés Masegosa, Thomas D. Nielsen, Antonio Salmerón	Final version of document

1 Executive summary

In this deliverable, we provide an extended and more general overview of the software library related to the implementation of the AMIDST modelling framework previously presented in Deliverable 2.1 [1].

In particular, we describe the main core components of the implemented framework, including the data structures used to develop the AMIDST models, the data source management functionalities associated with both learning and inference engines defined in WP3 and WP4, respectively, as well as the HUGIN AMIDST API ensuring the interaction between AMIDST toolbox and HUGIN software.

2 Introduction

The AMIDST modelling framework [1] is based on probabilistic graphical models (PGMs) that consist of two main components: a qualitative component in the form of a graphical model encoding conditional independence assertions about the domain being modelled, and a quantitative component consisting of local probability distributions adhering to the independence properties specified in the graphical model.

In AMIDST, we focus on two particular types of PGMs, namely, *Bayesian networks* and *dynamic Bayesian networks*.

Bayesian networks (BNs) [2,3] are widely used PGMs for reasoning under uncertainty. Formally, let $X = \{X_1, \ldots, X_n\}$ denote the set of n stochastic random variables defining a specific domain problem. A BN defines a *joint probability distribution* $p(X)$ in the following form:

$$p(X) = \prod_{i=1}^{n} p(X_i|Pa(X_i))$$

where $Pa(X_i) \subset X \setminus X_i$ represents the so-called *parent variables* of X_i. BNs are graphically represented by a directed acyclic graph (DAG). Each node, labelled X_i in the graph, is associated with a factor or conditional probability table $p(X_i|Pa(X_i))$. Additionally, for each parent $X_j \in Pa(X_i)$, the graph contains one directed edge pointing from X_j to the *child* variable X_i.

Figure 2.1 shows an example of a BN model including five variables. A conditional probability is associated to each node in the network describing its conditional probability distribution given the set of its parents in the network, so that the joint distribution factorises as:

$$p(X_1, \ldots, X_5) = p(X_1)p(X_2|X_1)p(X_3|X_1)p(X_4|X_2, X_3)p(X_5|X_3)$$

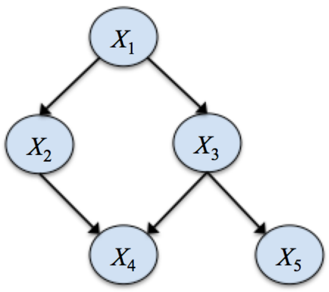

Figure 2.1: Example of a BN model with five variables.

A BN is called *hybrid* if some of its variables are discrete while some others are continuous. In the AMIDST modelling framework [1], we specifically consider *conditional linear Gaussian (CLG) BNs* [4–6], where the local probability distributions of continuous variables are specified as CLG distributions and where discrete variables can only have discrete parents. Therefore, in addition to univariate distributions, and depending on the type of the parent and child variables, i.e., continuous or discrete, the following conditional distributions can be distinguished:

- *Multinomial* | *Multinomial*: the discrete child follows an independent multinomial probability distribution for each configuration of its discrete parents.
- *Multinomial* | *Normal*: the continuous child is distributed as an independent conditional Gaussian distribution for each configuration of its discrete parents.
- *Normal* | *Normal*: the continuous child follows a CLG distribution, i.e., the mean parameter of its Gaussian distribution is a linear combination of its continuous parents, while the variance is a fixed independent parameter.
- *(Multinomial, Normal)* $\rightarrow$ *Normal*: for each configuration of the discrete parents, the continuous child follows an independent conditional Gaussian distribution depending on its continuous parents. In this case, the mean parameter of the Gaussian distribution of the continuous child is expressed as a different linear combination of the continuous parents for each configuration of the discrete parents, and the variance of this Gaussian can also be different.

The second type of PGM that was considered in AMIDST modelling framework is the dynamic Bayesian network (DBN) [7], which is basically used to model domains that evolve over time by representing explicitly the temporal dynamics of the system. DBNs can be then readily understood as an extension of standard static BNs to the temporal domain. In fact, similarly to static BNs, the problem is modelled using a set of stochastic random variables, denoted X^t, with the main difference that variables are indexed here by a discrete time index t.

In general, DBNs can model arbitrary distributions over time. In AMIDST modelling framework [1], we especially focus on the so-called *two-time slice DBNs* (2T-DBNs). 2T-DBNs are characterised by an *initial model* representing the initial joint distribution of the process and a *transition model* representing a standard BN repeated over time. This kind of DBN model satisfies both *the first-order Markov assumption* and *the stationary assumption.* The first-order Markov assumption ensures that knowing the present makes the future conditionally independent from the past, i.e., $p(X^{t+1}|X^{1:t}) = p(X^{t+1}|X^t)$, while the stationary assumption entails that changes in the system state are time invariant or time homogeneous, i.e., $p(X^{t+1}|X^t) = p(X^t|X^{t-1})\ \forall t \in \{1, \ldots, T\}$.

In a 2T-DBN, the transition distribution is represented as follows:

$$p(\mathbf{X}^{t+1}|\mathbf{X}^{t}) = \prod_{X^{t+1}\in\mathbf{X}^{t+1}} p(X^{t+1}|Pa(X^{t+1}))$$

where $Pa(X^{t+1})$ refers to the parent set of X^{t+1} in the transition model, which can be variables either at the same or the previous time step. Figure 2.2 shows an example of a graphical structure of a 2T-DBN model. For instance, we have $Pa(X_1^{t+1}) = X_1^t$.

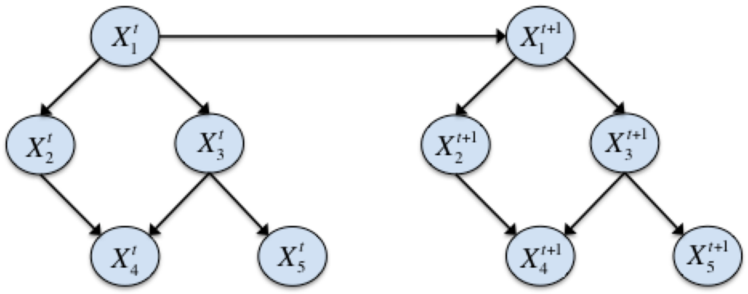

Figure 2.2: An example of a BN structure corresponding to a 2T-DBN.

Moreover, hidden or latent variables can be considered such as in the widely used hidden Markov models (see Deliverable 2.1 [1], Section 3.3.1), and Kalman or switching Kalman filter models (see Deliverable 2.1 [1], Section 3.3.2).

Along this deliverable, we present an overview of the software library implementation of the AMIDST toolbox, covering the methodological developments related to the above-described models and concepts, the data source management functionalities associated with both learning and inference engines, as well as HUGIN AMIDST API ensuring the interaction between AMIDST toolbox and HUGIN software. Recall that AMIDST toolbox is an open source, entirely new software developed from scratch within this project, while the HUGIN AMIDST API is an extension of the COTS HUGIN software.

The structure of the deliverable is as follows: Section 3 gives a general overview of the core components of the framework, including data structures for variables, graphs, Bayesian networks, dynamic Bayesian networks, key distributions such as multinomial and conditional linear Gaussian distributions represented in both standard form and as exponential families. Section 4 provides a description of the considered database functionalities that will be used by AMIDST learning and inference algorithms. Section 5 presents the functionalities defined for transforming AMIDST models to and from HUGIN software. Finally, Section 6 rounds the document off with conclusions.

3 Data structures

An overview of the data structures implemented in the AMIDST toolbox is illustrated in Figure 3.1. These data structures basically define the main components that will be used afterwards for implementing the AMIDST learning and inference algorithms. As we previously mentioned, in the AMIDST toolbox, we focus on two specific instantiations of PGMs, namely, a static Bayesian network (BN component) and a two time-slice dynamic Bayesian network (2T-DBN component).

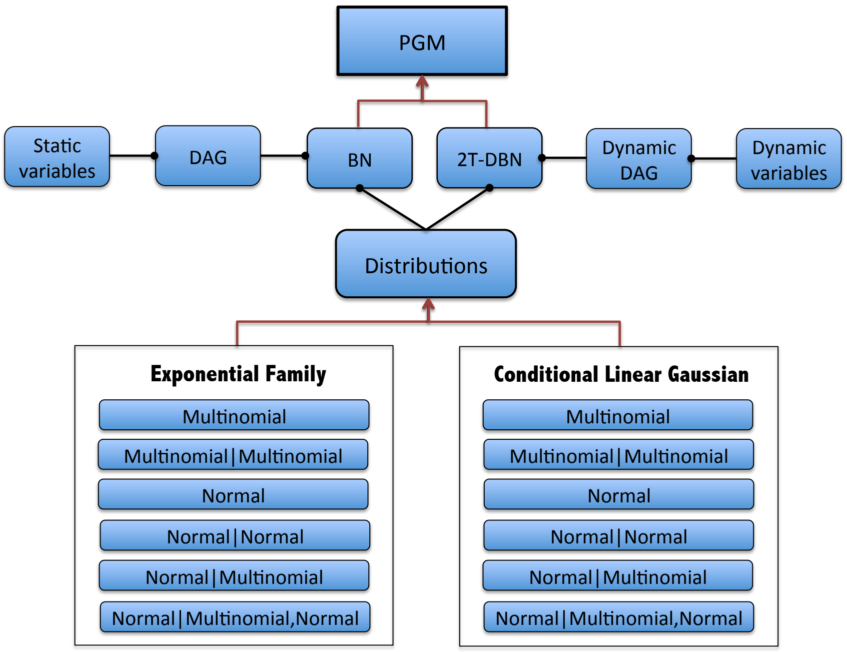

Figure 3.1: Illustration of AMIDST toolbox data structure components. Nomenclature: The boxes in the figure represent software components (sets, possibly singletons, of classes), a rounded-arc going from X to Y indicates that Y 'uses/references' X, and an arc with an arrow from X to Y implies inheritance.

In what follows, we briefly define each component and how it can be used in AMIDST toolbox through providing some code excerpts.

3.1 Static variables

Static variables consist of a list of objects of type Variable that are used later to build a static BN. Each static variable is characterized by its name, ID, the state space type, the distribution type (i.e., multinomial or normal), as well as if it is observed or not.

Note that observed static variables are initialised using the list of attributes (that are already parsed from the dataset or specified by the user), then hidden static variables could be afterwards specified by the user.

The following source code example shows how to define a set of four static observable variables and a single hidden variable:

```
DataStream<DataInstance> data =
          DataStreamLoader.loadFromFile("datasets/staticData.arff");
StaticVariables variables = new StaticVariables(data.getAttributes());
Variable A = variables.getVariableByName("A");
Variable B = variables.getVariableByName("B");
Variable C = variables.getVariableByName("C");
Variable D = variables.getVariableByName("D");
Variable H = variables.newMultionomialVariable("HiddenVar",
                         Arrays.asList("TRUE", "FALSE"));
```

3.2 Directed acyclic graph (DAG)

A directed acyclic graph (DAG) defines the BN graphical structure over a list of static variables, such that the dependence relationships between the variables are established through the definition of the parent set for each variable.

The following source code example shows how to build a DAG over the previously defined set of static variables, such that the hidden variable is set as a parent of all the remaining variables:

```
DAG dag = new DAG(variables);

dag.getParentSet(A).addParent(H);
dag.getParentSet(B).addParent(H);
dag.getParentSet(C).addParent(H);
dag.getParentSet(D).addParent(H);

System.out.println(dag.toString());
```

The last line converts the resulting dag into a String object, then prints it to the standard console. We obtain:

```
DAG
A parent sets: {HiddenVar}
B parent sets: {HiddenVar}
C parent sets: {HiddenVar}
D parent sets: {HiddenVar}
HiddenVar parent sets: {}
```

3.3 Bayesian network (BN)

As mentioned before, a static BN consists of two components: a graphical structure (defined by the DAG component) and conditional probability distributions of each variable given the set of its parents (defined by the Distributions component). Thus, given a DAG, the BN is defined through initialising the distribution of each variable according to its type and the type of its parent set. After this step, the set of parents of each variable becomes unmodifiable.

The following brief code fragment shows how to define a BN using a previously build `dag`. It automatically checks the distribution type of each variable and its corresponding parents to uniformly initialise the Distributions objects such as multinomial, normal, CLG, etc. (see Section 3.7).

```
BayesianNetwork bnet = BayesianNetwork.newBayesianNetwork(dag);
System.out.println(bnet.toString());
```

Similarly to DAG, the resulting `bnet` can be converted into a String object then printed to the standard console. We have:

```
Bayesian Network:
P(A [MULTINOMIAL] : HiddenVar [MULTINOMIAL], ) follows a Multinomial|Multinomial
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
P(B [MULTINOMIAL] : HiddenVar [MULTINOMIAL], ) follows a Multinomial|Multinomial
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
P(C [NORMAL] : HiddenVar [MULTINOMIAL], ) follows a Normal|Multinomial
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
P(D [NORMAL] : HiddenVar [MULTINOMIAL], ) follows a Normal|Multinomial
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
P(HiddenVar [MULTINOMIAL]) follows a Multinomial
[ 0.5, 0.5 ]
```

3.4 Dynamic variables

Dynamic variables consist of a list of objects named allVariables and temporalClones of type `Variable`, that are used to build dynamic Bayesian networks. Each dynamic variable is characterized by its name, ID, the state space type, the distribution type (i.e., multinomial or normal), and if it is observed or not.

In order to represent the variables in a previous time step (needed when defining the dynamic DAG), we use the concept of *temporal clone* variables, which are copies of the real main variables but refer to the previous time step. For instance, X^{t-1} is codified as the *temporal clone* of variable X^t. Hence, in our data structures, the time index t is not explicitly represented for a dynamic variable, but implicitly considered with the use of *temporal clones*.

The list of observable dynamic variables is initialised using the list of Attributes (that are already parsed from the dataset or specified by the user), then hidden variables can be also added by the user. Next, temporal clones are automatically created through invoking the `getTemporalClone()` method.

The following source code example shows how to define a set of four dynamic observable variables and two hidden dynamic variables:

```
DataStream<DynamicDataInstance> data =
       DynamicDataStreamLoader.loadFromFile("datasets/dynamicData.arff");
DynamicVariables dynamicVariables = new DynamicVariables(data.getAttributes());
Variable A = dynamicVariables.getVariable("A");
Variable B = dynamicVariables.getVariable("B");
Variable C = dynamicVariables.getVariable("C");
Variable D = dynamicVariables.getVariable("D");
Variable ATempClone = dynamicVariables.getTemporalClone(A);
Variable H1 = dynamicVariables.newMultinomialDynamicVariable("Hidden1",
                         Arrays.asList("TRUE", "FALSE"));
Variable H2 = dynamicVariables.newMultinomialDynamicVariable("Hidden2",
                         Arrays.asList("TRUE", "FALSE"));
```

3.5 Dynamic directed acyclic graph (Dynamic DAG)

A dynamic directed acyclic graph (Dynamic DAG) defined over a list of dynamic variables. This component specifies the graph structure of a 2T-DBN by specifying the parent set for each dynamic variable at time $t > 0$ (note that the parent sets at time 0 are automatically specified).

The following source code example shows how to build a dynamic DAG over the previously defined set of dynamic variables:

```
DynamicDAG dynamicDAG = new DynamicDAG(dynamicVariables);

dynamicDAG.getParentSetTimeT(A).addParent(ATempClone);
dynamicDAG.getParentSetTimeT(B).addParent(H1);
dynamicDAG.getParentSetTimeT(B).addParent(H2);
dynamicDAG.getParentSetTimeT(C).addParent(H1);
dynamicDAG.getParentSetTimeT(D).addParent(H1);
dynamicDAG.getParentSetTimeT(D).addParent(H2);
dynamicDAG.getParentSetTimeT(H1).addParent(ATempClone);
dynamicDAG.getParentSetTimeT(H2).addParent(ATempClone);

System.out.println(dynamicDAG.toString());
```

The last line converts the resulting `dynamicDAG` into a String object, then prints it to the standard console. We obtain:

```
Dynamic DAG at Time 0
A has 0 parent(s): {}
B has 2 parent(s): {Hidden1, Hidden2}
C has 2 parent(s): {Hidden1, Hidden2}
D has 2 parent(s): {Hidden1, Hidden2}
Hidden1 has 0 parent(s): {}
Hidden2 has 0 parent(s): {}

Dynamic DAG at Time T
A has 1 parent(s): {ATempClone}
B has 2 parent(s): {Hidden1, Hidden2}
C has 2 parent(s): {Hidden1, Hidden2}
D has 2 parent(s): {Hidden1, Hidden2}
Hidden1 has 1 parent(s): {ATempClone}
Hidden2 has 1 parent(s): {ATempClone}
```

3.6 Two time-slice dynamic Bayesian network (2T-DBN)

Similarly to a static BN, a 2T-DBN (see Deliverable D2.1, Section 3.4 [1]) is defined using two main components: a graphical structure (defined by the Dynamic DAG component) and conditional probability distributions of each dynamic variable given the set of its parents (defined by the Distributions component). Thus, given a Dynamic DAG, the BN is defined through initialising the distributions of each dynamic variable at both time 0 and time T according to its type and the type of its parent set. After this step, the set of parents of each dynamic variable becomes unmodifiable.

This is brief code fragment showing the definition of a dynamic Bayesian network using the previously created dynamicDAG. It automatically looks at the distribution type of each variable and their parents to initialise the Distributions objects that are stored inside (i.e., Multinomial, Normal, CLG, etc). The parameters defining these distributions are correspondingly initialised.

The following brief code fragment shows how to define a 2T-DBN using a previously build dynamicDAG. It automatically checks the distribution type of each variable and its corresponding parents to uniformly initialise the Distributions objects such as multinomial, normal, CLG, etc. (see Section 3.7).

```
DynamicBayesianNetwork dynamicbnet =
      DynamicBayesianNetwork.newDynamicBayesianNetwork(dynamicDAG);
System.out.println(dynamicbnet.toString());
```

Similarly to dynamic DAG, the resulting dynamicbnet can be converted into a String object then printed to the standard console. We have:

```
Dynamic Bayesian Network Time 0:
P(A[MULTINOMIAL]) follows a Multinomial
[ 0.5, 0.5 ]
P(B[MULTINOMIAL]: Hidden1[MULTINOMIAL], Hidden2[MULTINOMIAL]) follows a Multinomial|Multinomial
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
P(C[NORMAL]: Hidden1[MULTINOMIAL], Hidden2[MULTINOMIAL]) follows a Normal|Multinomial
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
P(D[NORMAL]: Hidden1[MULTINOMIAL], Hidden2[MULTINOMIAL]) follows a Normal|Multinomial
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
P(Hidden1[MULTINOMIAL]) follows a Multinomial
[ 0.5, 0.5 ]
P(Hidden2[MULTINOMIAL]) follows a Multinomial
[ 0.5, 0.5 ]
```

```
Dynamic Bayesian Network Time T:
P(A[MULTINOMIAL]: ATempClone [MULTINOMIAL]) follows a Multinomial|Multinomial
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
P(B[MULTINOMIAL]: Hidden1[MULTINOMIAL], Hidden2[MULTINOMIAL]) follows a Multinomial|Multinomial
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
P(C [NORMAL]: Hidden1[MULTINOMIAL], Hidden2 [MULTINOMIAL]) follows a Normal|Multinomial
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
P(D[NORMAL]: Hidden1[MULTINOMIAL], Hidden2[MULTINOMIAL]) follows a Normal|Multinomial
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
Normal [ mu = 0.0, sd = 1.0 ]
P(Hidden1[MULTINOMIAL]: ATempClone[MULTINOMIAL]) follows a Multinomial|Multinomial
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
P(Hidden2[MULTINOMIAL]: ATempClone[MULTINOMIAL]) follows a Multinomial|Multinomial
[ 0.5, 0.5 ]
[ 0.5, 0.5 ]
```

3.7 Distributions

The Distributions component consists of the set of conditional probability distributions considered in the AMIDST toolbox. It currently includes both multinomial and normal distributions, and could be easily extended in the future to cover additional distribution types.

Note here that, in spite of the distinction between BN and 2T-BN, the distributions over both models could be defined in the same way, and thereby the parameter learning and inference algorithms could be also applied equally for both models. In particular, the Distributions component includes the set of conditional probability distributions considered in the AMIDST toolbox (the so-called Conditional Linear Gaussian distributions, as detailed in Deliverable 2.1 [1]). More precisely, both variables with multinomial and normal distributions are modeled, and the distribution of each variable, in either a BN or 2T-BN, is initialized and specified according to its distribution type and the distribution types of its potential parents. This consequently gives rise to the following different implemented probability distributions:

- Multinomial: a multinomial variable with no parents.

- Multinomial|Multinomial: a multinomial variable with multinomial parents.
- Normal: a normal variable with no parents.
- Normal|Normal: a normal variable with normal parents.
- Normal|Multinomial: a normal variable with multinomial parents.
- Normal|Multinomial,Normal: a normal variable with a mixture of multinomial and normal parents.

The case of a multinomial variable having normal parents is not considered yet in this initial prototype. It is planned to be included in future versions, although strongly restricted in inference and learning algorithms due to the methodological and computational issues previously commented in Deliverable D2.1 [1].

We also provide an implementation of all the above distributions in the so-called Exponential Family form, which ensures an alternative representation of the standard distributions based on vectors of natural and moment parameters.

The following brief code fragment shows the definition of the distribution for the multinomial variable A given its multinomial parent, and the normal variable C given also its multinomial parent.

```
Multinomial_MultinomialParents distA = bnet.getDistribution(A);
distA.getMultinomial(0).setProbabilities(new double[]{0.7, 0.3});
distA.getMultinomial(1).setProbabilities(new double[]{0.2, 0.8});

Normal_MultinomialParents distC = bnet.getDistribution(C);
distC.getNormal(0).setMean(0.15);
distC.getNormal(0).setSd(0.5);
distC.getNormal(1).setMean(0.24);
distC.getNormal(1).setSd(1);
```

4 Database management

This section covers the description of databases that will be used later by AMIDST learning and inference algorithms implemented in the toolbox. Figure 4.1 shows a high-level overview of the key components of the AMIDST toolbox. It illustrates mainly the different database functionalities and how they are connected to the core component PGM through both the Learning Engine and Inference Engine components. In what follows, we describe each of the database functionalities, along with a code excerpt containing a brief example, then we introduce briefly the Learning Engine and Inference Engine that will be presented in more details in Deliverable 3.2.

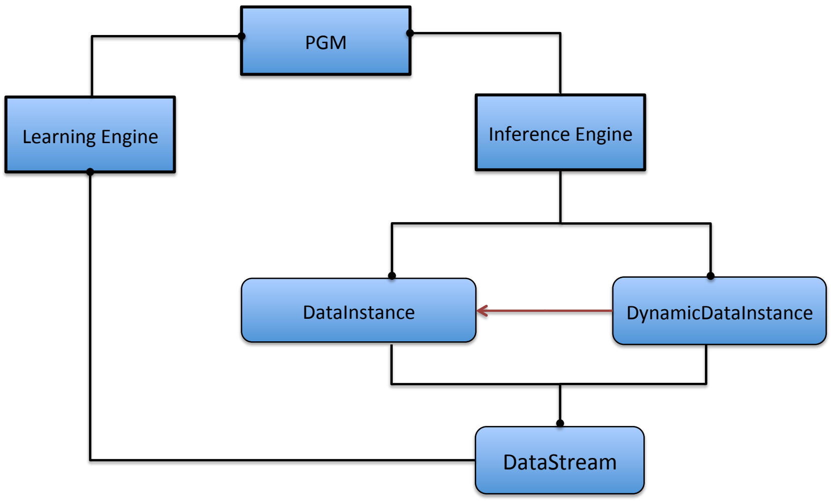

Figure 4.1: Illustration of the main database management functionalities and their connection with PGM, learning engine, and inference engine components.

4.1 DataStream

In the AMIDST framework, we consider a DataStream as a data source, i.e., a streaming data where records arrive at high frequency with no storage of historical data in memory. DataStream is connected to either DataInstance and DynamicDataInstance components.

The employed design is intended to support future users and developers of the AMIDST toolbox in the potential design and implementation of other database specifications; the only restriction being that new database components should implement the interface defined by the DataStream component.

4.2 DataInstance

The DataInstance component consists of a single class that represents a particular evidence configuration, such as the observed values of a collection of variables in a particular data row.

```
DataStream<DataInstance> data =
          DataStreamLoader.loadFromFile("datasets/staticData.arff");
```

4.3 DynamicDataInstance

The DynamicDataInstance component consists of two data rows, such that the first refers to the past while the second refers to the present. In addition to the attributes present in each data row, DynamicDataInstance could be characterised also with a TimeID and a SequenceID stored as additional attributes in the considered dynamic data.

```
DataStream<DynamicDataInstance> data =
     DynamicDataStreamLoader.loadFromFile("datasets/dynamicData.arff");
```

4.4 Learning Engine

The Learning Engine component consists of the implementations of the different learning algorithms for static and dynamic BNs, ensuring both structural and parameter learning. For structural learning, the AMIDST toolbox currently supports standard PC and parallel TAN algorithms by interfacing to the Hugin API (cf. Task 4.1). For parameter learning, a fully Bayesian approach is pursued in the AMIDST framework (cf. Task 4.2 and Task 4.4), which means that parameter learning reduces to the task of inference for which two approaches will be considered:, namely, variational message passing and expectation propagation.

More implementation details about these algorithms will be provided in Deliverable 3.2. Note also that the design of the Learning Engine is flexible in the sense that it easily accommodates potential future learning-based extensions, such as Bayesian learning based on importance sampling or maximum likelihood learning using the expectation maximization algorithm (see Section 3 in Deliverable 4.1 [8]).

4.5 Inference Engine

The Inference Engine component consists of the implementations of both variational message passing and expectation propagation algorithms for probabilistic graphical models with conjugate-exponential distribution families (see Section 3, Deliverable 4.1 [8]).

The different functionalities of the Inference Engine component are ensured in AMIDST toolbox through tailored exponential family implementations of the standard distributions that are part of the AMIDST framework (such as the conditional linear Gaussian distributions). More details about this component will be provided in Deliverable 3.2.

5 HUGIN AMIDST API

The Hugin link component consists of the functionalities implemented to link the AMIDST toolbox with the HUGIN software. This connection is primarily ensured by converting

HUGIN models into AMIDST models, and vice versa.

This component is extremely useful as it allows us to test and assess some of the implemented AMIDST functionalities within a well-established platform as HUGIN. For instance, a new inference algorithm implemented in AMIDST could be compared with some state-of-the-art algorithms included in HUGIN. In addition, the connection with HUGIN efficiently extends AMIDST toolbox by providing some extra functionalities, such as the use of parallel TAN for BN structural learning.

5.1 Converters from AMIDST to HUGIN format

This functionality addresses the conversion of a BN or a DBN from AMIDST to HUGIN. This conversion is done at “object-level”, which is far more efficient that if it would be done by converting the models to data files and, then, parsing them.

The following brief code fragment shows how to convert a BN and a DBN from AMIDST to HUGIN format, then store them in files. The files could be then accessed and opened using HUGIN software to visually check the created networks with the AMIDST toolbox.

```
%For a BN:
BayesianNetwork amidstBN = BayesianNetwork.newBayesianNetwork(dag);
Domain huginBN = BNConverterToHugin.convertToHugin(amidstBN);;
huginBN.saveAsNet("networks/huginNetworkFromAMIDST.net");

%For a DBN:
DynamicBayesianNetwork amidstDBN =
     DynamicBayesianNetwork.newDynamicBayesianNetwork(dynamicDAG);
Class huginDBN = DBNConverterToHugin.convertToHugin(amidstDBN);
huginDBN.setName("huginDBNFromAMIDST");
huginDBN.saveAsNet("networks/huginDBNFromAMIDST.oobn");
```

5.2 Converters from HUGIN to AMIDST format

This functionality addresses the conversion of a BN or a DBN from HUGIN to AMIDST. As before, this conversion is also done at “object-level”.

The following brief code fragment shows how to convert a BN and a DBN from HUGIN to AMIDST format, then store them in files. The files could be then accessed and used in AMIDST toolbox.

```
%For a BN:
ParseListener parseListener = new DefaultClassParseListener();
Domain huginBN = new Domain ("networks/asia.net", parseListener);
BayesianNetwork amidstBN = BNConverterToAMIDST.convertToAmidst(huginBN);
BayesianNetworkWriter.saveToFile(amidstBN, "networks/asia.bn");

%For a DBN:
DefaultClassParseListener parseListener = new DefaultClassParseListener();
ClassCollection cc = new ClassCollection();
String file = new String("networks/CajamarDBN.oobn");
cc.parseClasses (file, parseListener);
String[] aux = file.split("/");
String fileName = aux[aux.length-1];
String modelName = fileName.substring(0,fileName.length()-5);
Class HuginDBN = cc.getClassByName(modelName);
DynamicBayesianNetwork amidstDBN = DBNConverterToAmidst.convertToAmidst(HuginDBN);
DynamicBayesianNetworkWriter.saveToFile(amidstDBN, "networks/CajamarDBN.bn");
```

6 Conclusion

This document described the software library implementation of the AMIDST framework including the implementation details of the main core components, namely, data structures and database functionalities, as well as the presentation of HUGIN AMIDST API interface ensuring the connection between AMIDST toolbox and HUGIN software.

In Deliverable 3.2, more implementation details of AMIDST toolbox will be presented especially those related to the inference and learning engines.

References

[1] Borchani, H., Fernández, A., Gundersen, O.E., Hovda, S., Langseth, H., Madsen, A.L., Martínez, A.M., Martínez, R.S., Masegosa, A., Nielsen, T.D., Salmerón, A., rmo, F.S., Weidl, G.: The AMIDST modelling framework – Initial draft report (September 2014) Deliverable 2.1 of the AMIDST project, `amidst.eu`.

[2] Jensen, F., Nielsen, T.: Bayesian networks and decision graphs. Second edn. Springer Publishing Company, Incorporated (2007)

[3] Pearl, J.: Probabilistic Reasoning In Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1988)

[4] Lauritzen, S.L.: Propagation of probabilities, means, and variances in mixed graphical association models. Journal of the American Statistical Association **87**(420) (1992) 1098–1108

[5] Lauritzen, S.: Graphical Models. Oxford University Press (1996)

[6] Lauritzen, S., Jensen, F.: Stable local computation with conditional Gaussian distributions. Statistics and Computing **11**(2) (2001) 191–203

[7] Dean, T., Kanazawa, K.: A model for reasoning about persistence and causation. Computational Intelligence **5**(3) (1989) 142–150

[8] Borchani, H., Langseth, H., Nielsen, T.D., Salmerón, A., Madsen, A.L.: Progress report on the software development (December 2014)