The iOOBN framework: technical details

Md. Samiullah David Albrecht Ann Nicholson Faculty of Information Technology,
Monash University, Victoria, Australia
msam34@student.monash.edu, david.albrecht@monash.edu, ann.nicholson@monash.edu

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abstract

The construction of Bayesian Networks (BNs) to model large-scale real-life problems is challenging. One approach to scaling up is Object Oriented Bayesian Networks (OOBNs). These provide modellers with the ability to define classes and construct models with a compositional and hierarchical structure, enabling reuse and supporting maintenance. In the OO programming paradigm, a key concept is inheritance, the ability to derive attributes and behavior from pre-existing classes, which enables an even higher level of reusability and scalability. However, inheritance in OOBNs has yet to be fully defined and implemented. Here we present iOOBN, a tool which provides fully defined inheritance for OOBNs. We provide guidance on modelling in iOOBN, describe our prototype implementation with an existing BN software tool, Hugin, and demonstrate its applicability and usefulness via a case study of re-engineering an existing large complex dynamic OOBN.

The iOOBN framework: technical details

I. Introduction

In real-world applications the ability to reason under uncertainty is critical for making decisions. Bayesian decision networks (BNs) [30], [15] are a powerful tool for performing and supporting many forms of uncertain reasoning, including monitoring, prediction, diagnosis, risk assessment and decision support. Their usefulness as a mature modelling technology is demonstrated by the extremely wide range of areas to which they have been applied, including (with single examples only, for reasons of space): medicine [9], education [10], agriculture [22], ecology and environmental management [2], biosecurity [3], surveillence [28], miliary [11], weather forecasting [6] and software engineering [33].

Bayesian decision networks can be built "by-hand" using elicitation methods to capture expert knowledge, from models in the literature, by automated learning if data is available, or by combinations of these. Particularly when much or all of the model is built by-hand, as the complexity of problem increases, BN modelling methods struggle to scale up. The resultant large complex BNs are difficult to visualise and hard for the domain experts and decision makers to understand, reducing the acceptance and subsequent use of the model.

From the late 1990s, researchers started to develop theories and techniques to scale-up BN modelling. These included versions of well-known techniques for handling complexity, such as dividing the problem into subparts and then combining the BN models for the subproblems, and re-using BN model previously built and validated for another application. These techniques include object-oriented BNs (OOBNs) [20], [5], PRM [18], OOPRM [34], generalised decision-graphs [29], BN fragments [25], varieties combining probabilistic relational models and objects, such as module networks [32], probabilistic relational models and plate models [14], multi-entity BNs (MEBNs) [24], idioms [12], and template-based representations [19].

OOBNs are inspired by object oriented principles from software engineering (e.g. [7]), where sub-parts of the overall model are represented in classes, which contained both nodes and objects, which are instances of other classes, giving a composite and hierarchical structure; these key concepts were first introduced by Koller and Pfeffer [20]. Connections between objects is strictly limited to define input and output interface nodes, providing the OO concepts of encapsulation, abstraction and information hiding. Thus a modeller may embed an object in the larger model without knowing about its details, and the decision maker can view all or parts of the model at different levels of abstraction, aiding understanding and acceptance. The use of classes also supports maintenance of the models, as a change (e.g. updating the model parameters when new data becomes available) need only be made once, in the class, and the change is automatically propagated to all the objects of that class, in any number of OOBNs. Other advantages of OOBNs include: supporting the building of large OOBNs in parallel by multiple modellers, who only need to agree on the class interfaces; provide modularity which limits the scope of changes and reduces the chance of a model change introducing errors; facilitating the design of both temporal[1] and spatial models.

The power and efficacy of OOBNs have been demonstrated in their application across a similar range of application domains as BNs including medicine, environmental management, agriculture, reliability engineering and mechatronics (see [21], Ch5). However, despite their advantages, OOBNs comprise only a small proportion of all BNs. Possible explanations for this include: that their modelling overhead is not justified for smaller, simpler and one-off models; they are not supported in most BN software packages; there are no modelling guidelines or knowledge engineering methodologies for OOBNs.[2]. They also have the major limitation that the key OO concept of inheritance has not been fully defined and implemented in any existing BN software tool.

In the OO paradigm, inheritance is the ability to derive attributes and behavior from pre-existing classes, which enables a greater level of reusability and scalability. In current OOBN modelling, every new class is a separate entity, regardless of its similarities to existing classes. Incorporating inheritance into an OOBN framework means allowing classes to be "extended", with new classes – called sub-classes – to be defined in terms of "inheriting" certain elements from the original class, along with the differences. This should allow the re-use of already constructed network segments, as well as supporting maintenance.

The original description of OOBNs [20] did not provide for the expressiveness of dynamic object construction and changing, and was not implemented by the authors. The first implementation of OOBNs was in the 2003 version of Hugin [26], a widely used and (with Netica) the longest-established commercial BN software tool, but still does not include any implementation of inheritance. The OOBN framework presented in [5] did provide a limited form of inheritance (only allowing a subclass to change the interface and the hidden structure).

Probabilistic relational models (PRM) [18] are an alternative to OOBNs that arose in the early 2000s inspired by relational database theory, relational algebra and relational logic programming. To provide full access to the class components in order to facilitates modellers to model large and complex Bayesian network applications, PRM contains reference slots to establish relations among classes. However, these reference slots violate the encapsulation and data hiding mechanism of the OO paradigm and hence introduces challenges in decomposing large applications that are developed across a group of

[1] Hugin implements dynamic Bayesian networks, used for explicit reasoning over time, using OOBNs

[2] Unlike for BNs, where guidelines and methodologies include [8], [27]

modellers. While the original PRM did not include inheritance, OOPRM [34] extends the PRM framework by introducing OO concepts such as inverse reference slot, interfaces, inheritance and polymorphism. OOPRM has the same issues as PRM with reference slots, while inverse reference slots also violate encapsulation, and it is less flexible to extension, modification, decoupling, and decomposition. Moreover, in OOPRM, the reference slot chains and inverse reference slot chains make modification and re-use far more complex, as the modeller needs to consider how a class is embedded in the whole system, when making even a simple modification to a single class. In addition, while they allow compact representation of relationships between classes that will be instantiated with multiple objects, they do not provide the utility and decision nodes that allow BNs (and OOBNs) to be used for decision making. Finally, while OOPRM has been implemented in research software [13], it is not available in any commerical modelling tool, and there seem to be few real-world OOPRM models described in the literature.

The original formulation of OOBNs is free from all above mentioned limitations and has been most widely used in practice, due to its implementation in Hugin. In this paper, we present a tool, which we call *iOOBN*, which extends the Hugin OOBN implementation with a full treatments of inheritance and its associated OO concepts, namely object formation, instantiation, polymorphism, dynamic maintenance, and type checking.

The rest of the paper is organised as follows. In Section II, we present iOOBN, a tool with a full treatment of inheritance in OOBNs. In Section III we provide guidance (although not yet a full methodology) for knowledge engineering models within the iOOBN framework, and describe our implementation of a prototype of the iOOBN framework in Section IV. While in this paper we deal only with discrete variables, the framework can be extended to continuous variables in a straightforward way. In Section V, we re-engineer an existing real-world, complex, dynamic OOBN (for an ecological management response model) into the iOOBN framework, demonstrating the applicability, usefulness and power of iOOBNs. Finally, in Section VI we conclude with some ongoing and future research directions.

II. The iOOBN Tool

Here we describe all the iOOBN components to provide proper treatment of all the features of OO paradigm, especially inheritance. We extend the definition of OOBNs used in [17] and [21], and implemented in Hugin BN, also following that terminology and notation. We assume standard Bayesian decision network definitions of chance nodes (representing random variables), decision nodes and utility nodes, directed arcs, conditional probability tables (CPTs), and utility tables.

A. Basic Components

An iOOBN *class* consists of nodes, *objects* which are instances of classes, and edges. There are two types of classes, abstract and concrete. *Concrete classes* are fully parameterised and thus can be 'flattened' into a standard Bayesian decision network and compiled; these correspond to classes as defined in previous formulisations of OOBN. In contrast, *abstract classes* are not fully parameterised, i.e. do not have fully specifed CPTs and/or utility tables, and therefore cannot be compiled, i.e. cannot be used for reasoning.

As for standard Bayesian decision networks, all the nodes in an iOOBN class have a name and are either chance nodes, decision nodes or utility nodes. *Chance nodes* represent random variables. They have a set of states, a CPT, and are represented by an oval shape. *Decision nodes* represent the possible actions that a decision maker must choose between. They have a set of actions and are represented by a rectangular shape. The final type of node is the *utility node*, which has an associated utility table (representing a utility function over its parent chance and decision nodes) and is represented by a diamond shape.

In an iOOBN class (as in standard OOBNs) there are four types of edges or links: ordinary, information, precedence, and referential links. *Ordinary links* are the standard edges in a Bayesian decision network, represented by a single solid-line arrow; they can be from a chance node to a chance node, from a chance node to a utility node, or from a decision node to a utility node. An *information link* is represented by a dotted line arrow and is from a chance node to a decision node; it indicates when a chance node must be observed before a decision is made. A *precedence link* is also represented by a dotted line arrow and is from a decision node to a decision node; it represents the sequence order between decisions. Finally, *referential links* are used to connect nodes to nodes within objects and are represented by double dotted lines. For two nodes to be linked by a referential link, they must be either both chance nodes with the same states, or both decision nodes with the same actions. When an iOOBN is flattened into a BN, nodes that are joined by referential links are represented by a single node. The edges within an iOOBN must be such that it will flatten out to a valid BN, that is, directed, acyclic graph.

Classes (and hence objects) in iOOBN (as in standard OOBNs) have three types of nodes: input, output and embedded nodes. The *input* and *output nodes* of a class, called its *interface*, can be either chance nodes or decision nodes, while the *embedded nodes* of a class can be either chance nodes, decision nodes or utility nodes. A class can also contain objects, instances of other classes. An object is connected to a node in the class in which it sits (called its *encapsulating* class), if there is a referential link between the node and an input node in the object's interface. A node in the interface can only have one referential link to a node outside the object. In addition, there may be ordinary links from an object's output nodes to embedded or output nodes outside the object. If an input node of an embedded object (of a concrete class) is not connected via referential link, it must have a default CPT (as per [15]); if connected via a referential link, the CPT of the connected node overrides the default CPT.

We denote an input node with a dotted circle, output node with a double-lined circle, embedded chance, decision and utility nodes with shaded circles, shaded rectangles and shaded diamonds, respectively. For textual distinction of input, output and embedded basic nodes, we have used italic, bold face

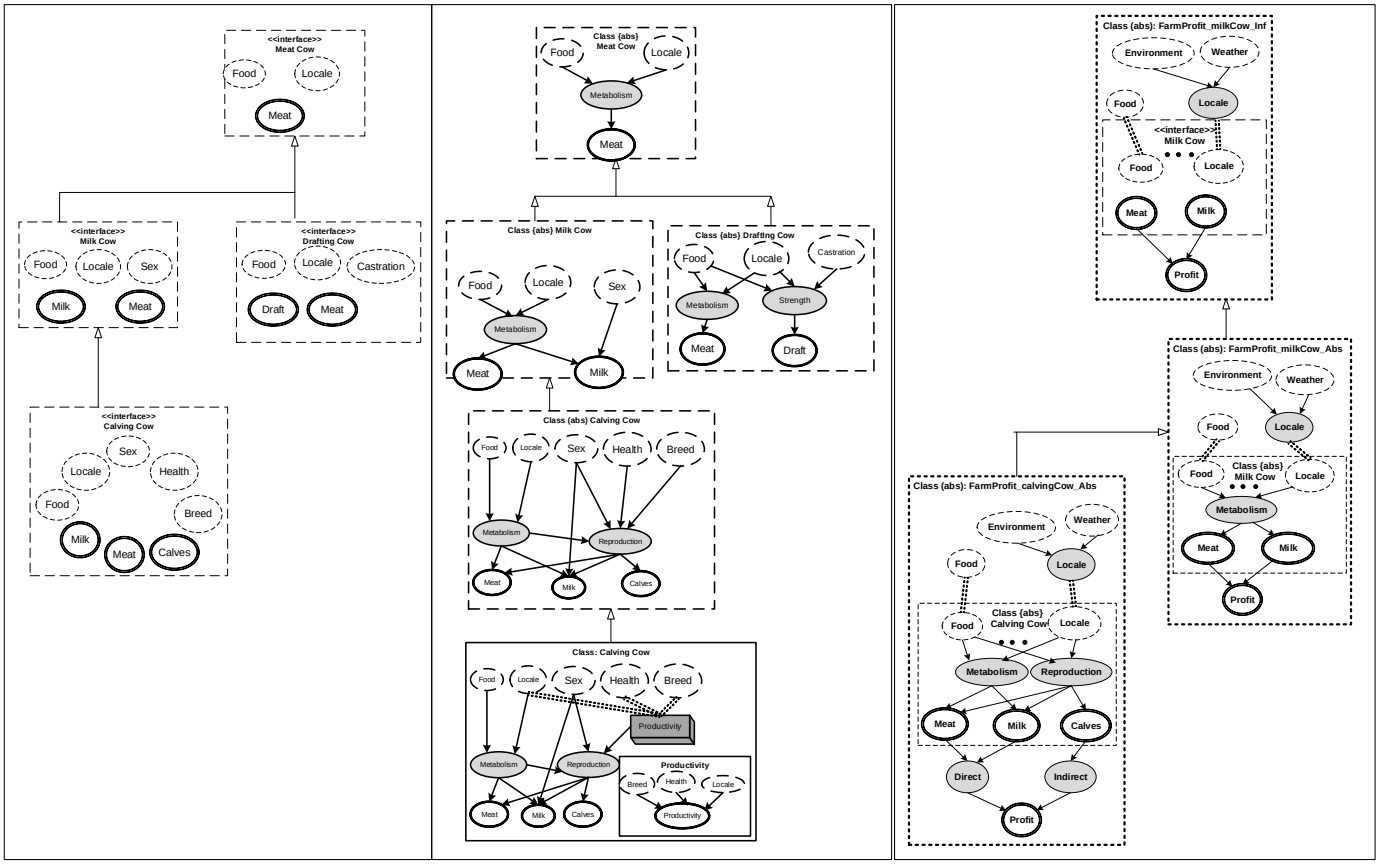


Fig. 1. Example interface and class hierarchy for the OMD livestock example [iOOBN framework components and polymorphism].

and underlined font, respectively. To describe objects, we have used underlined-italic font.

In addition to concrete and abstract classes, iOOBN also includes an *interface* as a separate data structure, containing only input and output nodes, without any parameters (i.e. these nodes have empty CPTs). For every abstract or concrete class, there is always exactly one associated interface (though the modeller might not choose to explicitly use it). An interface acts as a placeholder in the enscapulating class during the model design process, allowing the model builder to use sub-models built previously, or being built simultaneously by other modellers, without having to understand the inner details of that sub-model. Hence, when building a large, complex iOOBN, the modellers need only to agree on the mutual interfaces of classes beforehand and can combine their works later. The aim of the model building process is to produce a fully specified iOOBN, that is an object (instance) of the top level class that can be compiled and run. At some stages during that process, all interfaces and abstract classes must be replaced by an object, an instance of a concrete class, having the same interface nodes.

The specification of abstract classes, and interfaces, which provide the OO features of abstraction, encapsulation and generalization, are key components of iOOBN that differ from previous OOBN formulations. Limiting the connections between classes to the interface nodes provides encapsulation, a way to deny unauthorized access to the information and structure of the encapsulated sub-models. We note however that the information hiding within an iOOBN is not "pure"; evidence can be entered for any chance node in the iOOBN, however deeply it is embedded, and the posteriors (for chance nodes) and expected utilities (for decision nodes) computed by the inference algorithm are available as the iOOBN model "outputs".[3]

For illustration purposes, we use the livestock farming of the Old McDonald (OMD) [23] example with necessary extensions. In OMD, the farm owner has several types of cows ("milk", "meat", "drafting", "calving"), augmented with decision and utility nodes to model farming decisions based on profit making. Fig. 1 (slot-1 and slot-2) shows a UML-type representation of the OMD hierarchy in iOOBN. Here we describe the basic components in this iOOBN example; in the following section we will describe the class and interface hierachy elements of the example.

Consider the concrete Calving Cow class in Fig. 1 (2nd slot, bottom): it has five input nodes ($Food$, $Locale$, Sex, $Health$ and $Breed$), two embedded nodes (Metabolism and Reproduction), an embedded object *Productivity*, which is an instance of the *Productivity* class, and three output nodes (**Meat**, **Milk** and **Calves**). There are referential links (undirected double dotted lines) from three of those input nodes

[3]This is analogous to I/O in OO programming, which can occur in any object in the overall software.

($Food$, $Locale$ and $Breed$) to the input nodes of the Productivity object, and a single ordinary link (solid line) from the embedded object (the output **Productivity** node, not shown) to the embedded Reproduction node. Because it is a concrete class, all the CPTs must be fully specified (although this is not shown in the figure for reasons of space). The abstract Calving Cow class has the same input and output nodes as the concrete Calving Cow class; it has a different internal structure – this will be explained below. There is also an interface Calving Cow class (1st slot), connected via a dotted line with open arrowhead going from the abstract class to its corresponding interface class (following standard OO UML notation).

B. Inheritance and polymorphism

Inspired by the OO principle of inheritance, in iOOBN both classes and interfaces, denoted sub-types, can inherit structure and parameters (analagous to inheriting attributes and/or behaviors in OO programming) from another entity (class or interface, respectively), denoted the super-type. A sub-type entity is a variation, usually thought of as a specialisation, that includes changes to structure and/or parameters.

We define a *sub-interface* as inheriting all the input and output nodes from its super-interface, and which may have any number of additional input or output nodes. Similarly, a *sub-class* inherits all the input and output nodes from its *super-class*, and may have any number of additional input or output nodes. The sub-class inherits all of the internal structure of the super-class (embedded nodes and objects), and may also contain additional nodes and links. The interface nodes of the sub-class are then a superset of those of the super-class. Because all input nodes in concrete classes must have default CPTs specified, there is no problem replacing an object of a super-class with a sub-class. The inheritance relationship is denoted with a solid line with open arrow directed into the super-class, following standard UML inheritance notation.

Consider again the OMD example in Fig. 1. The highest level class is the abstract Meat Cow class (2nd slot, top), because all cows in this farm example can be used for meat production. The Meat Cow class in turn has two (abstract) sub-classes (Meat Cow and Drafting Cow). Both these sub-classes inherit the Meat Cow super-classes input nodes (*Food* and *Locale*) and output node (**Meat**), but have additional interface nodes (*Sex* and **Milk** for Milk Cow class, *Castration* and **Draft** for Drafting Cow). Drafting Cow has an additional interface node, *Strength*, an important attribute for a drafting cow, as well as an additional output node (**Draft**). Similarly, we can see that the previously described Calving Cow abstract class is a sub-class of Milk Cow, and has a more complex structure: two new input nodes (*Health* and *Breed*) and additional output node (**Calves**) and different internal structure around the new embedded node *Reproduction*. We can see that the concrete Calving Cow is itself a sub-class of the abstract Calving Cow class, with exactly the same interface nodes, but with a different internal and more complex structure, specifically around an additional embedded object *Productivity*. Similarly, there is an inheritance hierarchy between the interfaces Meat Cow, Milk Cow and Calving Cow, with same solid line with open arrow notation; naturally, these correspond to the different interfaces found in the class hierarchy.

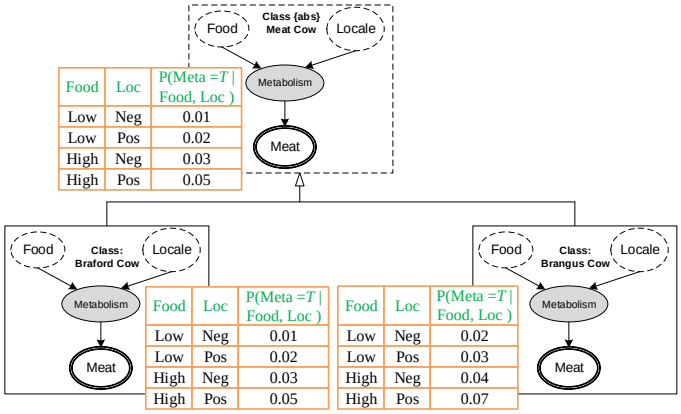


Fig. 2. Changing CPTs of inherited nodes in the sub-classes

A crucial aspect of OO inheritance, which must apply here in iOOBN as well, is that the classes remain "backward" compatible; this means that an encapsulated object can be replaced by an object of any sub-type (direct, or via the hierarchy) and the resultant iOOBN is still a valid iOOBN – nothing "breaks", and when made fully concrete, can be compiled and run (i.e. used for reasoning). This gives iOOBN another prominent feature of the OO paradigm, namely "Polymorphism", which allows an object to take many forms if and when required. Unlike OO programming, this replacement of an object by another that is a sub-type, cannot be done at runtime; rather, it must be done during the modelling process. We can simply model this special case as a set of classes that differ only in their embedded objects, as a hierarchy of classes, as for example depicted in Fig. 1 (slot-3), where the FarmProfit_milkCow_Inf super-class contains an embedded interface object (Milk Cow interface), its sub-class FarmProfit_milkCow_Abs contains an embedded object of the corresponding Milk Cow abstract class, and in turn the FarmProfit_calvingCow_Abs sub-class (derived from the class FarmProfit_milkCow_Abs) contains an embedded object of the abstract sub-class Calving Cow.[4]

There are further analogies we can make with OO programming. Where an iOOBN sub-class inherits certain elements from its super-class, any changes to those elements, such as changes in the domain or type of node(s), are a form of *overriding*. iOOBN is a strongly typed framework where each entity has an associated type, which allows us to perform type checking to ensure that the types of interface nodes are preserved in the sub-class. Moreover, when a super-class object can be replaced by an object of its sub-class, then this is a form of *type casting*.

[4]Note that we have considered adding an additional element to iOOBN, namely to have "variants" of a class which differ only in the embedded object and all the rest of the class remains the same (including the referential links to the encapsulated object). This feature would be utilised in applications where a script is used to generate different versions of the model for different purposes (as in the WGR case study described in Section V), allowing a form of runtime binding. However this remains a possible future extension to iOOBN.

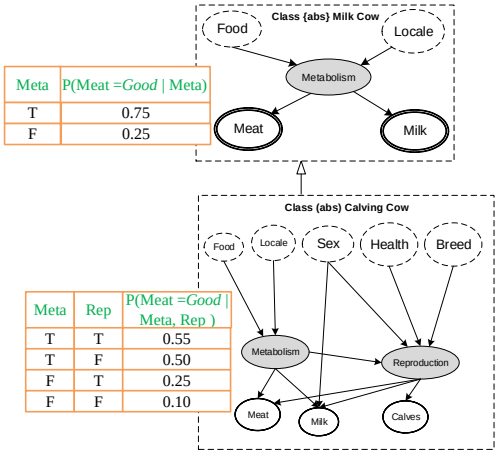


Fig. 3. Adding links and nodes within sub-classes

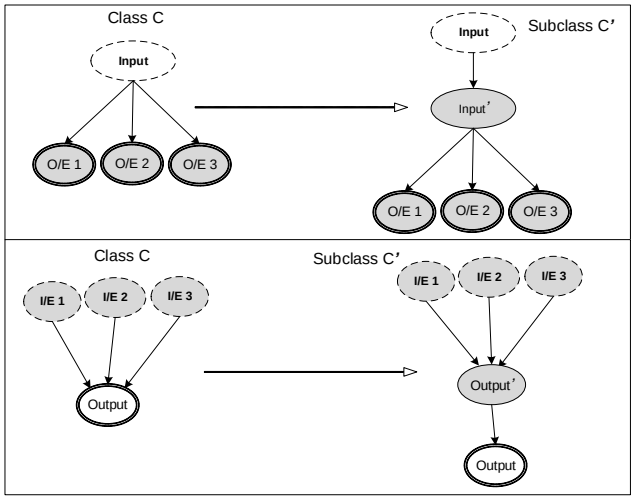


Fig. 4. Schematic diagram showing the model change required to effect a change in the states of an input (above) or output (below) nodes.

III. Modelling within the iOOBN framework

The BN modelling process, in practice, is usually iterative and incremental, e.g., [25], [21], particularly when building a model partially or completely "by hand", rather than learning from data. So the modelling process can be seen as a sequence of changes to the BN: adding or deleting nodes or edges, changing the name or states of a node, changing some or all of a CPT, and so on. Also, some of these changes necessarily generate others; for example, adding a new state to a node means changing the CPT of both the node itself, and also the CPTs of any child nodes.

When building an OOBN, as well as the ordinary BN modelling changes, there are additional possibilities for modelling steps, such as: embed an object within a class, create a new class, change the referential links to an embedded object, and so on. When modelling with ordinary OOBNs, as implemented in Hugin, if the modeller wants to create a new class that has common elements with an existing class - for example, if s/he had created a Meat Cow class, and wanted to make new Milk Cow, this would involve copying the Meat Cow class, giving it a new name, then making changes in the new Milk Cow class itself. When modelling complex problems, the number of classes can grow quite large (as we see in our case study in Section V), making it difficult for the modeller to keep track of the differences and similarities of the various new classes and increasing the chance of inadvertent errors and inconsistencies. In addition, if at any point the modeller wants to change some part of the model that is common to more than one class the modeller often needs to make the same change in more than one class. For example, in ordinary OOBN modelling, changing the state space of the Locale node in the Meat Cow node would suggest the same change should be made in the Milk Cow class (and any others).[5]

Inheritance provides an approach for solving these issues. Using inheritance we can record the connection between the variations of a class, and allows a change in one class to flow through to other relevant classes. Thus, using the iOOBN framework, the modelling steps are: to create a new sub-class (e.g. Meat Cow); give it a new name (e.g. Milk Cow); then make a sequence of changes to the new sub-class itself. The iOOBN system must then first check that any given change is valid, and then keep track of those changes. Thus the sub-class is internally represented by what it inherits from its parent, i.e., where it is identical, and then by the set of changes that override elements of the parent class.

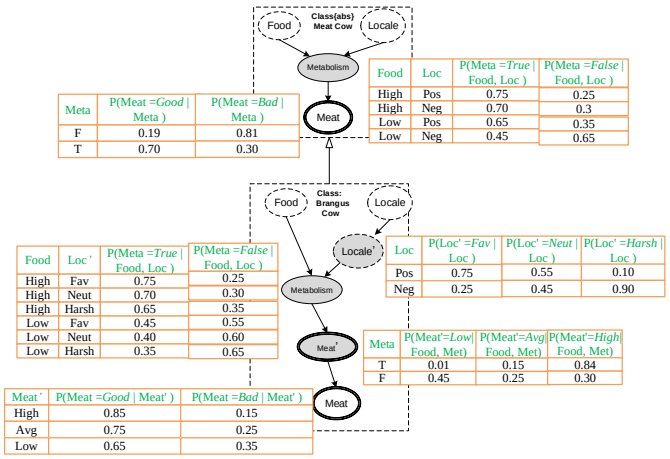
The simplest form of sub-class is one where only the parameters of the sub-class are changed and all structural elements remain the same; Fig. 2 gives an example of this case, with two sub-classes of the Meat Cow class created to represent two different species of cows, where the CPT of the Metabolism node has been changed from the parent Meat Cow class, to represent the different metabolic rates of **Bradford** and **Brangus** cows. Note that the figure shows the resultant new CPT, not the internal representation of the inherited CPT plus changed elements.

Any kind of nodes – input, output and embedded – can be added to a sub-class. This would normally involve the addition of edges, otherwise the change will result in the degenerate case having a disconnected node. Embedded nodes can be deleted, but the definition of inheritance – where the interface of a class and sub-class must be identifical – forbids the deletion of input or output nodes. However, if an input node is not relevant in the sub-class, it is possible to delete the links from an input node to its children, disconnecting it from the rest of the structure.

Fig. 3 illustrates an example where the sub-class has involved the creation of new nodes as well as edges, to model the reproduction process in our OMD example.

Another modelling step the iOOBN system must handle is a change in the states of a node. The first case for this is a change to the states for an embedded node in a class; this is straightforward and handled in exactly the same way as a change the CPT of a node.

[5]Of course ordinary OOBN still provide an efficiency in that a change to a class will be pushed out to any object created from that class. In constrast, when using sub-networks in the GeNIe software [1], the modeller can make copies of the same network fragment, but to make a change in that fragment, s/he has to change it in all sub-network copies, a tedious process.



Meta	P(Meat =*Good* | Meta)	P(Meat =*Bad* | Meta)
F	0.19	0.81
T	0.70	0.30

Food	Loc	P(Meta =*True* | Food, Loc)	P(Meta =*False* | Food, Loc)
High	Pos	0.75	0.25
High	Neg	0.70	0.3
Low	Pos	0.65	0.35
Low	Neg	0.45	0.65

Food	Loc'	P(Meta =*True* | Food, Loc)	P(Meta =*False* | Food, Loc)
High	Fav	0.75	0.25
High	Neut	0.70	0.30
High	Harsh	0.65	0.35
Low	Fav	0.45	0.55
Low	Neut	0.40	0.60
Low	Harsh	0.35	0.65

Loc	P(Loc' =*Fav* | Loc)	P(Loc' =*Neut* | Loc)	P(Loc' =*Harsh* | Loc)
Pos	0.75	0.55	0.10
Neg	0.25	0.45	0.90

Meta	P(Meat'=*Low* | Food, Met)	P(Meat'=*Avg* | Food, Met)	P(Meat'=*High* | Food, Met)
T	0.01	0.15	0.84
F	0.45	0.25	0.30

Meat'	P(Meat =*Good* | Meat')	P(Meat =*Bad* | Meat')
High	0.85	0.15
Avg	0.75	0.25
Low	0.65	0.35

Fig. 5. Type Casting: Type Changing of inherited Input-Output Nodes in sub-classes

Input: Class C
Output: Subclass C'
Create the subclass C'
Add a new node I' that has the new set of states
Add a link from I to I'
foreach *link e from I to its children (child) in C* **do**
  Remove e
  Add a new link from I' to *child*
  Update CPT of *child*
end
Construct a CPT for I'
return *subclass C'*

Algorithm 1: Modelling steps when adding states to an input node

A second and more interesting case is when the desired change is either to the states of an input node, or to the states of an output node of a class, **C**. By definition, this is not allowed, as the interface of a class and its sub-class must be the same. However we can make the desired change by modifying the non-interface structure of the class; adding a duplicate node with the different type and connecting it to the interface node, leaving the interface unchanged. Thus, the algorithm for the modelling steps to in effect change the type of an input node **I**, without changing the actual interface, is shown in Algorithm 1 while the steps to change the type of an output node, **O** in a class C are shown in Algorithm 2.

Fig. 4 gives a schematic representation of the changes effected by Algorithms 1 and 2. Since in the input node case, its children can be either embedded or output nodes, we combine our notation (shading plus bolded outline) for these child nodes, while for the output node case, its parents can be either embedded or input nodes, we combine the shadded and dotted outline notations for these parent nodes.

In Figure 5, we provide an example of this state change, when the states {"Pos", "Neg"} for the node *Locale*, are changed to {"Fav", "Neut", "Harsh"}, and the states {"Good", "Bad"} for the node **Meat** are changed to {"High", "Avg", "Low"}.

Input: Class C
Output: Subclass C'
Create the subclass C'
Add a new node O' that has the new set of states
Add a link from O' to O
foreach *link e to O from its parents (parent) in C* **do**
  Remove e
  Add a new link from *parent* to O'
  Create the CPT for O'
end
Construct a CPT for O
return *subclass C'*

Algorithm 2: Modelling steps when adding states to an output node

As these examples show, there are many possible changes that can be made when creating a new sub-class, and it will be both domain dependent and at the discretion of the modeller, as to how many changes to make in a single inheritence step. There will also be decisions to be made about whether to create multiple levels of subclasses, or to create a single level with multiple sub-classes. We expect that in some cases, an iOOBN class hierachy created incrementally might well be suboptimal and that regular re-engineering might be beneficial; however that is certainly an area for future investigation, after we have more experience in modelling within iOOBN.

IV. Prototype Implementation

We have developed a prototype iOOBN implementation using the Java programming language and the Hugin BN software API [16]. This prototype is based on a newly defined language NET++, an extended version of NET used in Hugin, which converts the iOOBN definitions of nodes, edges, classes (abstract and concrete) and interfaces to equivalent NET language in the HUGIN OOBN definition. The HUGIN engine is then used to perform all the standard belief updating and other Bayesian decision network operations.

A. Mechanisms to carrying and propagating/ maintaining changes

1) Steps of processing in iOOBN framework: Modellers can design BN systems in iOOBN with inheritance facility which reduces effort of making new classes from scratch. Actually, the OO facilities (specially inheritance) allow the modellers to create new classes from existing classes.

2) Creating new classes/interfaces/hierarchies: In the hierarchy shown in the figure, the interface "MeatCow is the root component and needs to be modelled from scratch. However, the interfaces "MilkCow and "DraftingCow can be created by extending (copying the nodes from another interface and adding additional nodes) "MeatCow.

Similarly, the abstract class "MeatCow can be created by implementing (copying all input, output nodes and adding additional embedded nodes with required edges in) the interface "MeatCow.

Furthermore, to create abstract classes "MilkCow and "DraftingCow we can extend (copying whole structure i.e. the

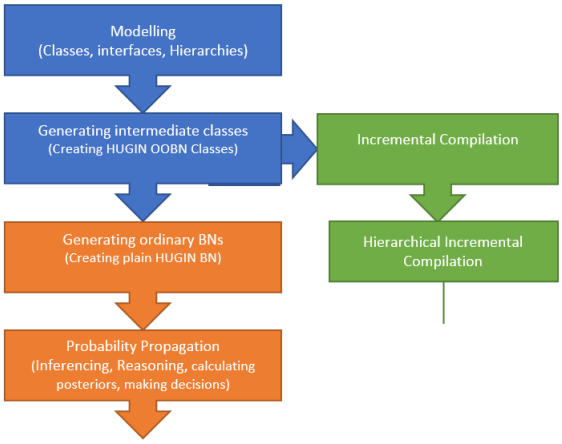


Fig. 6. Steps of processing iOOBN framework

nodes-edges and adding additional required edges, or nodes and edges in) the abstract class "MeatCow.

Hence, there are two special operations (1) Extend (2) Implement. We can add super() to use particular elements of a parent class/interface to create new ones.

3) Generating intermediate classes: We generate HUGIN OOBN classes for each of the concrete iOOBN classes. The iOOBN concrete classes implement several interfaces and extends at most one abstract or concrete iOOBN class, where the interfaces and classes might recursively extend or implement other classes or interfaces, respectively.

Hence, to generate a meaningful HUGIN OOBN class, we start from the top of the hierarchy (generated by a post analysis of the generated iOOBN interfaces, abstract classes and concrete classes). Then a BFS (Breadth First Search) strategy is followed to generate HUGIN OOBN classes level-by-level from the hierarchy tree.

B. Local stand-alone system

1) Current version (single phase compilation): The steps described above (modelling in IOOBN and generating HUGIN OOBN classes) are followed in the current pilot version i.e. Local stand-alone system.

2) Propagation/maintenance of changes in the hierarchy: The biggest problem of this system is, it requires changing from top to bottom of the whole hierarchy tree in a model each time we compile a model built in IOOBN.

3) Efficient version (Only recompiling changed classes): Keeping track of the classes that require changes and locating them in the hierarchy. Then marking all classes those reside in the branches leading from that particular location down to the bottom level.

This will save huge computational effort and unnecessary conversion in the intermediate class generation process.

4) More efficient version (Two phase compilation): Introducing hierarchical, incremental and shareable compilation techniques with minimal updating required.

5) Re-engineering WGR (Western Grassland Reserve) project: As a proof-of-concept case study, we re-engineered an existing real-world dynamic OOBN model i.e. WGR, implemented in Hugin, into the IOOBN framework. This model was developed as an environmental decision support tool for the so-called Western Grassland Reserve Project, modelling grassland species composition and how this might change under different management regimes.

C. iOOBN: documentation and workflow diagram

1) The data structure "Meta-Node Table: A Meta-Node Table (analogous to symbol table of compiler theory) is created which is a mapping between constructs (interfaces, abstract classes and concrete classes) name and their corresponding meta-node representation.

This table is used by loader-linker of iOOBN system to generate intermediate HUGIN OOBN codes.

2) The data structure "Meta-Node Tree and Forest: A list of meta-nodes is connected based on their dependency in a (imaginary) tree format. The dependencies are defined by the inheritance mechanism used in defining classes, abstract classes and interfaces as shown in the first figure.

There can be more than one tree i.e. a list of tree representing a particular model i.e. a forest (a non-empty list of trees) representing the hierarchical relations among the interfaces, classes (abstract/concrete).

This tree and forest is a handy tool for propagating the update/changes from parent meta-nodes to child meta-nodes. In particular, if a modeller changes definition of any class/interface which has been inherited by other classes/interfaces, then the changes needs to be reflected in the inheriting constructs. These trees and forest are the best optimized tool (so far I know) to pushing the changes from parents to children and so on.

3) Link between "Meta-Node Table and "Meta-Node Tree/Forest: While creating an iOOBN interface or class (abstract/concrete) straight forwardly i.e. without extending any class or implementing any interface, generating HUGIN OOBN class is quite simple or effortless.

However, if any class/interface inherits other interfaces or class then we immediately generate Hugin OOBN class for the inheriting ioobn class/interface and store in the meta-node structure. The inheriting process is simply a copy operation from inherited meta-node to inheriting meta node.

The "Meta-node Table can be extended to represent the tree structure by just adding parent and child class/interface references (index and/or name).

A sample table can be viewed as following:

4) Executable generation:

- Pre-processing
  1) Removing comments and unnecessary codes
- Compiling
  1) Parsing files to get meta node information
  2) Creating Meta-nodes for each file
  3) Completing meta-nodes based on dependency and inheritance
  4) Type checking and consistency checking
  5) Generating Hugin OOBN classes equivalent to the iOOBN class

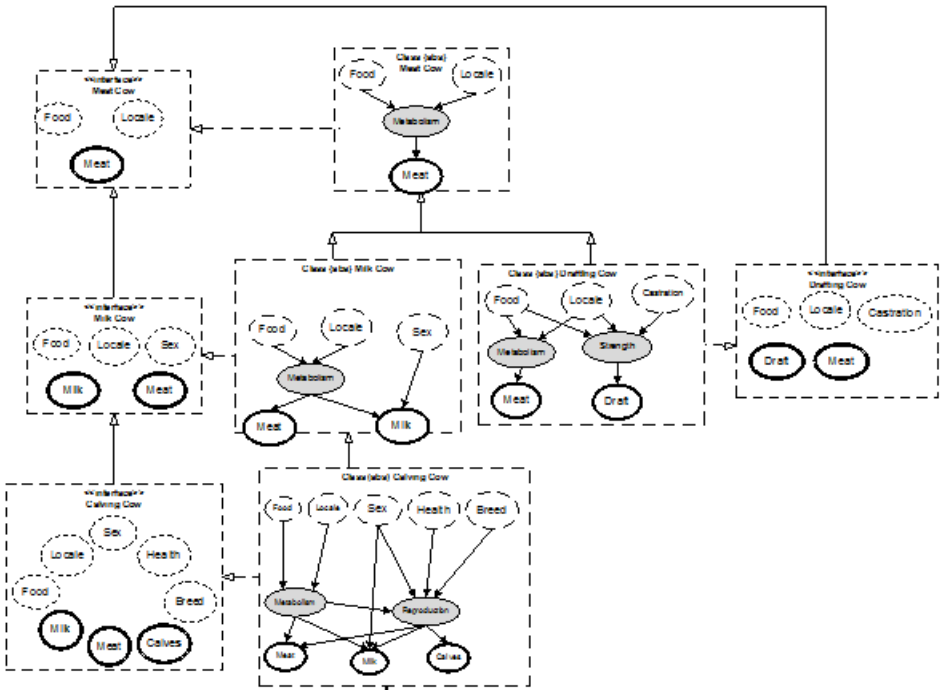


Fig. 7. Class hierarchy: Farm example

Par class	Par Inf List
Type(inf/abs/con)	Unique Name
BN definition goes here Nodes Edges (with/out CPTs) Instances (mapping)	
Par Ref	Chd Ref

Fig. 8. Generic Meta node structure

Ind	Name	Meta Node	Children List	Parent List
0	MeatCowInf	NodeInstance1	1, 2, 4	-
1	MilkCowInf	NodeInstance2	3, 5	0
2	DraftingCowInf	NodeInstance3	6	0
3	CalvingCowInf	NodeInstance4	7	1
4	MeatCowAbs	NodeInstance5	5, 6	0
5	MilkCowAbs	NodeInstance6	7	1, 4
6	DraftingCowAbs	NodeInstance7	-	2, 4
7	CalvingCowAbs	NodeInstance8	-	3, 5
8	.	.	.	.
9	.	.	.	.
10	.	.	.	.

TABLE I
TABLE REPRESENTING META-NODE TREE

MeatCowAbs	DraftingCowInf
Abstract	DraftingCowAbs
Emb: Strength Edge(Strength|Food Locale Castra){} Edge(Draft |Strength){}	

MilkCowAbs	CalvingCowInf
Abstract	CalvingCowAbs
Emb: Reproduction Edge(ReProd|Meta Sex Health Breed){} Edge(Calves | ReProd){} Edge(Milk | Meta Sex Reprod){} Edge(Meat | Meta ReProd){}	

	MilkCowInf
Abstract	MeatCowAbs
Emb: Metabolism Edge(Metabolism |Food Locale){} Edge(Meat | Meta){}	

MeatCowAbs	MilkCowInf
Abstract	MilkCowAbs
Edge(Milk | Meta Sex){}	

Fig. 9. Example Meta Nodes for the Meta-tree(part-1)

6) Updating Meta-Node table based on class/interface definition

- Assembling
  1) This part is not yet used in our iOOBN framework
- Loading and linking
  1) The linker will arrange the pieces of OOBNs so that one BN can use instance of another.
  2) Maintaining consistencies of changes from parent to children constructs and change propagation for instances used in other classes

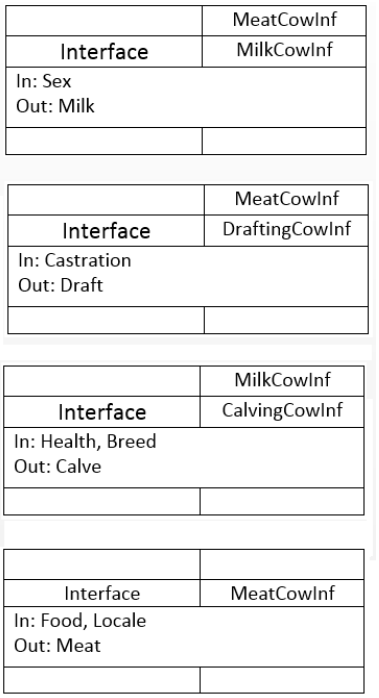


Fig. 10. Example Meta Nodes for the Meta-tree(part-2)

D. Implementing the system: iOOBN

We have developed a prototype implementation of our proposed framework using Java programming language and the Hugin Bayesian network software API. This prototype is based on a newly defined language NET++, an extended version of NET used in Hugin. This version of the framework converts the IOOBN framework definitions of nodes, edges, classes (abstract and concrete) and interfaces to equivalent NET language in the HUGIN OOBN definition. Then the HUGIN engine is used to perform belief updating and other Bayesian decision network operations.

The grammar for NET++ is given in a Table as a supplementary material.

V. Case Study

As a proof-of-concept case study, we re-engineered an existing real-world dynamic OOBN model, implemented in Hugin, in the iOOBN tool. The original OOBN model was developed as an environmental decision support tool for the so-called Western Grassland Reserve Project [35], modelling grassland species composition and how this might change under different management regimes.

A. Western Grassland Reserve Project

"WGR (Western Grassland Reserve) is dealing with the grassland vegetation management project to assist by a Bayesian network model. The fertile grasslands occupied by Melbourne city have been an attractive place for agricultural and settlement purpose for long time. As a consequence, substantial loss of natural habitats has been caused, and many animals, plants and vegetation communities are now listed as rare or threatened under the federal Environment Protection and Biodiversity Conservation (EPBC) Act 1999.

An agreement (Melbourne Strategic Assessment [MSA]) between Victorian State Government and The Australian Government has been sanctioned under the EPBC Act in 2010. Its focus is on EPBC listed (1) Three ecological communities (a type of grassland, a woodland and a wetland), (2) six plant species (two daisies, two orchids, a lily and a Rice-flower) and (3) four listed animal species (a frog, a bandicoot, a moth and a lizard).

For urban growth, the MSA permits specified areas of territory for these EPBC-listed species and communities to be cleared, along with the subsequent reservation and management of territory in conservation areas away from urban development (DPDC 2009, DELWP 2015).

Victoria is obliged to apply an adaptive management framework to the conservation areas, where clear goal for conservation outcomes are set, management options are trialled and monitored, and the results are communicated and translated into improvement management. Effective conservation management demands the ability to understand ecological change.

To be able to direct changes via a range of possible management actions, an intelligent choice is most likely to result when the manager is able to:

1) Predict the potential consequences of each action
2) Evaluate the desirability of the predicted consequences
3) Calculate the corresponding costs of each action

A good understanding of the ecosystem including its spatial and temporal dynamics and the range of possible responses is required to do these things accurately. For ages, such knowledge has been held and transmitted orally by indigenous people, hardly ever integrated into a cultural model of the decision theory and computer power, allowing the prediction and evaluation of complex outcomes.

Ecological management response models take as input

1) The initial status of the system, stated as response variables (e.g. the initial condition of the ecosystem, etc.), and
2) The environmental conditions the system will be subject to during the period of predictions (e.g. the expected climate conditions, a range of alternative management interventions, the potential threats and their impacts, etc.).

The models make predictions about the status of the response variables at the end of the prediction period using these inputs. The ingredient of the model is thus the causal relationship between the initial status and the environmental conditions. This relationship can be encoded using many different modeling approaches like:

1) Deterministic / rule-based models
2) Stochastic simulations such as Markov chain models
3) Bayesian Networks. (Plain BN, DBN, SBN, OOBN)

At the heart of ecological modelling is the trade-off between complexity and simplicity. Ecological systems are inherently complex and messy, while models are most easily built, tested, used and communicated when they are simple and sophisticated.

Prog	→ classStructure | abstractClassStructure | interfaceStructure
classStructure	→ 'class' className parentClass parentInterfaces '{' classElement* '}'
abstractClassStructure	→ 'abstract' 'class' className parentClass parentInterfaces '{' classElement* '}'
interfaceStructure	→ 'interface' interfaceName parentInterfaces '{' interfaceElement* '}'
interfaceElement	→ basicNode | attribute+ | classInstance
parentClass	→ 'extends' superClassName | ϵ
parentInterfaces	→ 'implements' interfaceNameList | ϵ
interfaceNameList	→ interfaceName (',' interfaceName)*
classElement	→ domainElement | attribute+ | classInstance
domainElement	→ basicNode | potential
classInstance	→ 'instance' instanceName ':' className '(' bindings ')' '{' instanceAttributes '}'
instanceAttributes	→ label | position | attribute
bindings	→ inputBindings | inputBindings ';' outputBindings | ';' outputBindings | ϵ
inputBindings	→ inputBinding (',' inputBinding)*
inputBinding	→ formalName '=' actualName
outputBindings	→ outputBinding (',' outputBinding)*
outputBinding	→ formalName '=' actualName
basicNode	→ 'node' nodeName '{' nodeAttribute* '}' | nodeType 'node' nodeName '{' nodeAttribute* '}' | 'decision' nodeName '{' nodeAttribute* '}' | 'utility' nodeName '{' nodeAttribute* '}'
nodeType	→ 'discrete' | 'continuous'
nodeAttribute	→ state | label | position | attribute | subtype
state	→ 'states' '=' '(' STRING* ')' ';'
label	→ 'label' '=' STRING ';'
position	→ 'position' '=' '(' xCoordinate yCoordinate ')' ';'
subtype	→ 'subtype' '=' 'boolean' ';' | 'subtype' '=' 'label' ';' | 'subtype' '=' 'number' ';' stateValues | 'subtype' '=' 'interval' ';' stateValues
stateValues	→ 'state_values' '=' '(' NUMBER* ')' ';'
potential	→ 'potential' '(' edgeInformation ')' '{' (potentialAttribute*) '}'
edgeInformation	→ childNodes | childNodes '|' parentNodes
childNodes	→ IDENTIFIER+
parentNodes	→ IDENTIFIER+
data	→ 'data' '=' '(' tuple ')' ';'
tuple	→ NUMBER | '(' tuple ')' | tuple NUMBER | '(' tuple ')' tuple
potentialAttribute	→ data | modelAttributes
attribute	→ attribName '=' attribValue ';'
modelAttributes	→ 'model_data' '=' stmt ';' | 'model_nodes' '=' '(' IDENTIFIER* ')' ';' | attribute
stmt	→ '(' stmt ')' | formula ',' stmt | formula | ϵ
attribValue	→ STRING | NUMBER | '(' NUMBER+ ')' | '(' IDENTIFIER+')' | '(' ')'
formula	→ expr
function_call	→ IDENTIFIER '(' ')' | IDENTIFIER '(' parameters ')'
parameters	→ parameter | parameters ',' parameter
parameter	→ formula
expr	→ boolExpr
boolExpr	→ sumExpr (('<' | '<=' | '>' | '>=' | '==' | '!=') sumExpr)*
sumExpr	→ productExpr (('-' | '+') productExpr)*
productExpr	→ primary (('/' | '*') primary)*
primary	→ literal | function_call | '(' formula ')'
literal	→ 'true' | 'false' | NUMBER | STRING | IDENTIFIER

TABLE II
CONTEXT FREE GRAMMAR FOR NPP (NET PLUS PLUS) LANGUAGE

This reflects the tension between traditional cultural models and scientific models. Scientific models are powerful and precise, but rigid only to explicit specification, and thus often lack the flexibility of cultural models. This is partly improved by the inclusion of elements of probability, uncertainty and randomness in the structure of scientific models; but despite these advances, scientific models can still only deal with what they are designed to deal with, and are thus biased in nature, and lacking in wisdom.

Hence, the consideration of variables for the model and how they are connected is crucial. Ideally, such a framework should help choosing right thing to do, allow change to be evaluated, and help learning from the experience. Ecologists and land managers have recognized this need for such a framework for decades (HollingClark1975, HobbsNorton1996, LindenMayer2008).

In recent times, Adaptive Management has been prominent as an organizing framework. It involves the implementation of several management systems that promise to achieve management objectives, monitoring and evaluation of the outcomes, synthesis of new knowledge to reduce uncertainty, and the alteration of management that incorporates new knowledge (walters1986, McCarthy2007, Morrison2012).

Implementation of adaptive management demands finer-level rationalization, including the ecosystem functions models, a set of quantitative goals, and a monitoring approach.

The Bayseian Network model describes in greater detail the species composition of the grassland, and how this composition is predicted to change under different management regimes. Within the adaptive management approach, the BN has been used for three main purposes:

1) To make predictions about the effects of management

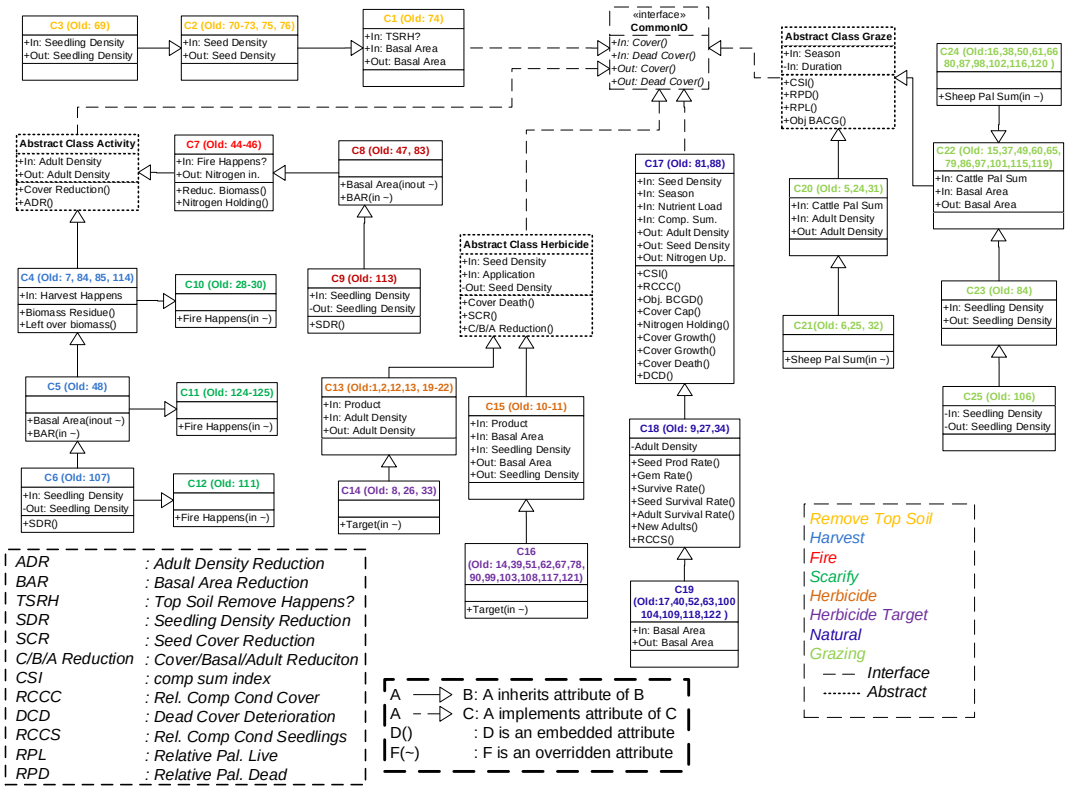


Fig. 11. Class hierarchy (machine learning approach)

(including cost-benefit considerations), allowing us to select a small set of promising options to trial in the field, from the vast array of possible management options.

2) To make our detailed assumptions about grassland ecology and management explicit, and open to criticism and improvement.
3) As a tool to help us learn, and a frame in which to house that learning. The BN parameters can be updated as we learn more about grasslands and their management, and strengthen our knowledge base

B. Original WGR implementation

In WGR, there is already a form of OOBNs proposed and available in the Hugin BN software implementation. Not only is it a well-designed and fully implemented working model, but it contains sufficient complexity to demonstrate what we hope to show are the advantages of I-OOBN (for example, multiple class structures that have similar elements, e.g. Themeda and other native grasses. By re-engineering (and re-implementing) using I-OOBN, we will compare the accuracy and efficiency of the I-OOBN version with original DOOBN version.

Ecological management response model presented original WGR project is complex and have a number of limitations with respect to object-oriented principles, including no elements of class inheritance. The limitations of the model other than inheritance are:

1) Lots of specific information is available (e.g. life history and germination requirements of many grassland species).
2) Lots of interventions are available.
3) Multiple scales of management.
4) Lots of expert opinion.

Knowledge extraction from the existing DOOBN WGR model is laborious and requires extensive computation of same repeated structures redundantly. Inferencing, decision making or generating reports from such enormous system is subject to efficiency and scalability. Moreover, extending such a complex and vast system is quite challenging. In exchange of such design, the model provided with respect to proposed IOOBN framework is extensible, reusable, scalable and efficient.

C. Re-engineering WGR

The WGR DOOBN contains 129 classes, with 96 containing only chances nodes with the same set of attributes distributed among the class (representing the growth cycle of different kinds of native and exotic grasses), and 33 classes that contained chance, decision and utility nodes containing other sets of attributes (representing the management strategies and their effect on the growth cycle). We re-engineered the classes in these two groups.

We have re-engineered WGR DOOBN using three methods, machine learning method, expert elicitation and combination of both.

1) Based on background knowledge
   - Based on managerial intervention types, their effects, lifespan of plants, type (exotic/weeds/grass/grown from plants/seed grown/C3/C4 and etc) of plants

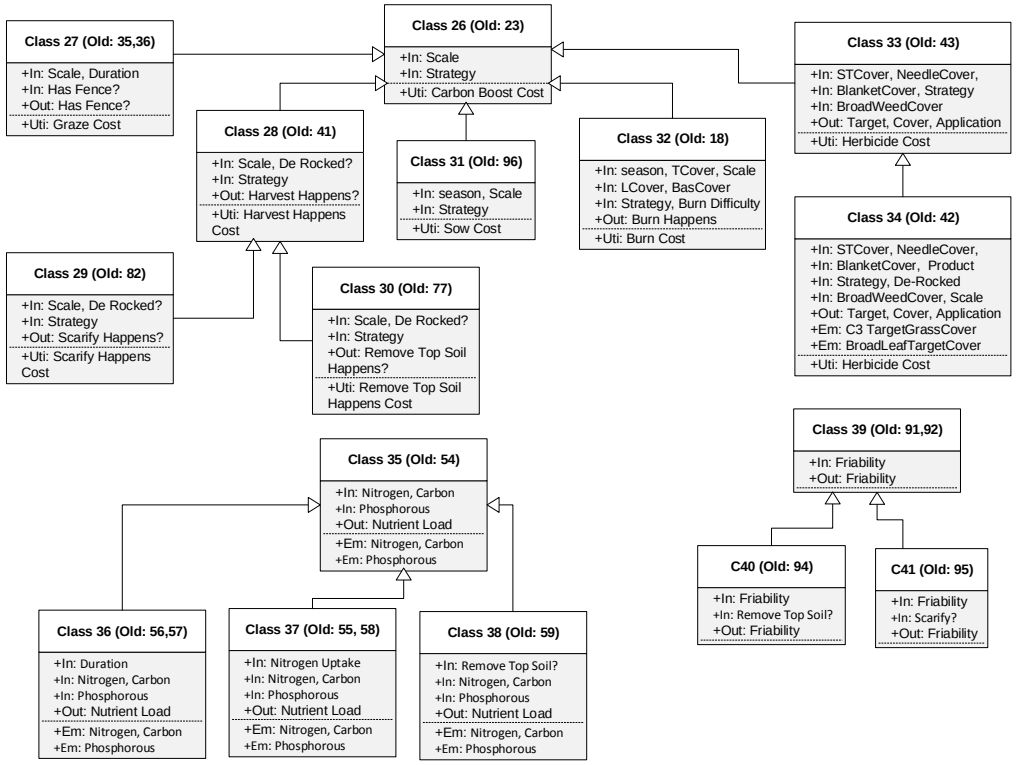


Fig. 12. Class (including utility and decision) hierarchy (machine learning approach)

2) Base on Machine learning and pattern mining:
   - It is a bottom-up approach. Each old class of OOBN forms a new class of IOOBN. Then all same graphical structure-oriented IOOBN classes are replaced by a new candidate class of IOOBN representing that group of classes. Afterwards, the groups/new classes are analyzed to find similarity among them.
   - If any K (¿= 2) number of classes have similar-partial similar structure then
   - If one class structure is a substructure of another class structure they will form a hierarchical structure with subclass-superclass relation
   - Else they will form a hierarchical structure with either a
   - common interface (no embedded node in the similar segment) or
   - common abstract class (at least one embedded node in the similar segment)

1) Base on Machine learning and pattern mining: First, the chance node only classes were classified into 25 groups based on their similarities and dissimilarities with respect to ecological and biological characteristics, with input from a domain expert involved in developing the original WGR DOOBN. This resulted in an interface node, 3 abstract classes and 25 concrete classes; these were in 4 distinct hierarchies with 23 derived classes (classes that have been constructed by inheriting properties of other classes or interfaces rather than from scratch).

The 33 more complex classes contained 13 that were distinct, dissimilar and large in size; no re-engineering into a class hierarchy was possible for these. From the remaining 20 classes, we re-engineered them into 16 concrete classes of three hierarchies with 13 derived classes. A UML class diagram for the re-engineered WGR model, along with a mapping of the original WGR classes into the IOOBN classes, is provided as a part of the supplementary material.

Overall, we reduced 129 classes into 1 interface, 3 abstract classes and 52 concrete classes, including 36 derived classes. This 57% reduction (129 classes down to 56 components) demonstrates that in large complex models there are indeed common elements that can be more compactly represented by a class hierarchy (rather than as a large number of unrelated classes that actually have many common elements, as in the original WGR OOBN).

2) Based on expert opinion and machine learning approach: In this method, we have taken into consideration the expert opinions to incorporate background knowledge and machine learning outcomes to build the hierarchical structure.

This resulted in 8 interfaces, 3 abstract classes and 25 concrete classes; these were in 4 distinct hierarchies with 23 derived classes (classes that have been constructed by inheriting properties of other classes or interfaces rather than from scratch).

The class hierarchy for remaining 33 more complex classes are same as machine learning method.

A UML class diagram for the re-engineered WGR model, along with a mapping of the original WGR classes into the IOOBN classes, is provided as a part of the supplementary material.

Overall, we reduced 129 classes into 8 interfaces, 3 abstract

Mapping of class names:

New column represents the class names/numbers in **reverse engineered version** and (**Old** + **Name**) columns represent the class names in **original WGR version**

Table with **Blue** Header represents constructs those are only available **IOOBN. Dark** rows indicates "Classes **not available** in the hierarchy".

New	Old	Name
1	74	Remove Top Soil 6
2	70	Remove Top Soil 2
	71	Remove Top Soil 3
	72	Remove Top Soil 4
	73	Remove Top Soil 5
	75	Remove Top Soil 7
	76	Remove Top Soil 8
3	69	Remove Top Soil 1
4	7	Blanket Harvest
	84	Sensitive Harvest 1
	85	Sensitive Harvest 2
	114	Tolerant Harvest
5	48	Moderate Harvest
6	107	Themeda Harvest
7	44	Killed Fire1
	45	Killed Fire2
	46	Killed Fire3
8	47	Moderate Fire
	83	Sensitive Fire
9	113	Tolerant Fire
10	28	Fragile Scarify 1
	29	Fragile Scarify 2
	30	Fragile Scarify 3
11	124	Tenacious Scarify 1
	125	Tenacious Scarify 2
12	111	Themeda Scarify
13	1	Annual Grass Herbicide 1
	2	Annual Grass Herbicide 2
	12	Broad Leaf Target Herbicide 1
	13	Broad Leaf Target Herbicide 2
	19	C3 Grass Herbicide
	20	C3 Grass Target Herbicide 1
	21	C3 Grass Target Herbicide 2
	22	C4 Grass Herbicide
14	8	Blanket Herbicide Target
14	26	Exotic Annual Herbicide Target
	33	Grain Herbicide Target
15	10	Broad Leaf Herbicide 1
	11	Broad Leaf Herbicide 2
16	14	Broad Weed Herbicide Target
	39	Hardy Native Herbicide Target
	51	Needle Herbicide Target
	62	Onion Herbicide Target
	67	Red Leg Herbicide Target
	78	Ruder Herbicide Target
	90	Sensi Native Herbicide Target
	99	Spear Herbicide Target
	103	ST Herbicide Target
	108	Themeda Herbicide Target
	117	Wallaby Herbicide Target
	121	Windmill Herbicide Target
17	81	Ruder Natural
	88	Sensi Native Natural
18	9	Blanket Natural
	27	Exotic Annual Natural
	34	Grain Natural
19	17	Broad Weeds Natural
	40	Hardy Native Natural
	52	Needle Natural
	63	Onion Natural
	68	Red Leg Natural
	100	Spear Natural
	104	ST Natural
	109	Themeda Natural
	118	Wallaby Natural
	122	Windmill Natural
20	5	Blanket Graze Cattle
	24	Exotic Annual Graze Cattle
	31	Grain Graze Cattle
21	6	Blanket Graze Sheep
	25	Exotic Annual Graze Sheep
	32	Grain Graze Sheep (*)
22	15	Broad Weed Graze Cattle
	37	Hardy Native Graze Cattle
	49	Needle Graze Cattle
	60	Onion Graze Cattle
	65	Red Leg Graze Cattle
	79	Ruder Graze Cattle
	86	Sensit Native Graze Cattle
	97	Spear Graze Cattle
	101	ST Graze Cattle
	115	Wallaby Graze Cattle
	119	Windmill Graze Cattle
23	105	Themeda Graze Cattle
24	16	Broad Weed Graze Sheep
	38	Hardy Native Graze Sheep
	50	Needle Graze Sheep
	61	Onion Graze Sheep
	66	Red Leg Graze Sheep
	80	Ruder Graze Sheep
	87	Sensit Native Graze Sheep
	98	Spear Graze Sheep
	102	ST Graze Sheep
	116	Wallaby Graze Sheep
	120	Windmill Graze Sheep
25	106	Themeda Graze Sheep
26	23	Carbon Boost
27	35	Graze Intervention Cattle
	36	Graze Intervention Sheep
28	41	Harvest Intervention
29	82	Scarify Intervention
30	77	Remove Top Soil Intervention
31	96	Sow Intervention
32	18	Burn Intervention
33	43	Herbicide Intervention Target
34	42	Herbicide Intervention
35	54	Nitrogen Carbon Boost
36	56	Nutrient Graze Cattle
	57	Nutrient Graze Sheep
37	55	Nutrient Fire
	58	Nutrient Natural
38	59	Nutrient Remove Top Soil
39	91	Soil Graze Cattle
	92	Soil Graze Sheep
40	94	Soil Remove Top Soil
41	95	Soil Scarify
42	3	Basal Adult Cover Graze (BACG)
43	4	Biomass Natural
44	64	Plant Intervention
45	53	Nui Adder
46	89	Sensitive Native Sow
47	93	Soil Natural
48	110	Themeda Plant
49	112	Themeda Sow
50	123	Basal Cover Growth Death (BCGD)
51	126	Biomass Summaries
52	127	Main
53	128	EnvValue
54	129	Nutrient Harvest [Found Blank]

Constructs	Name
Interface	CommonIO
Abstract Classes	Activity
	Graze
	Herbicide

Fig. 13. Old vs New class name mapping

classes and 52 concrete classes, including 36 derived classes. This 51% reduction (129 classes down to 63 components) demonstrates that in large complex models there are indeed common elements that can be more compactly represented by a class hierarchy (rather than as a large number of unrelated classes that actually have many common elements, as in the original WGR OOBN) with necessary background knowledge and more interpretable representation.

This kind of more compact representation also facilitates understanding of the model, and can obviously reduce the complexity of the associated documentation. Moreover, extending such a complex model is quite challenging, especially if the original BN modellers and/or the domain experts who build the original version are no longer available. Thus, this re-engineering case study suggests that the IOOBN framework can be useful for producing more efficient, reusable, extensible and scalable complex BN models.

D. Comparing two versions: Original vs. Reengineered

The WGR DOOBN contains 129 classes, with 96 containing only chance nodes with the same set of attributes distributed among the class (representing the growth cycle of different kinds of native and exotic grasses), and 33 classes that contained chance, decision and utility nodes containing other sets of attributes (representing the management strategies and their effect on the growth cycle). We re-engineered the classes in these two groups.

First, the chance node only classes were classified into 25 groups based on their similarities and dissimilarities with respect to ecological and biological characteristics, with input from a domain expert involved in developing the original WGR DOOBN. This resulted in 8 interface nodes, 3 abstract classes and 25 concrete classes; these were in 4 distinct hierarchies with 23 derived classes (classes that have been constructed by inheriting properties of other classes or interfaces rather than from scratch).

The 33 more complex classes contained 13 that were distinct, dissimilar and large in size; no re-engineering into a class hierarchy was possible for these. From the remaining 20 classes, we re-engineered them into 16 concrete classes of three hierachies with 13 derived classes. A UML class diagram for the re-engineered WGR model, along with a mapping of the original WGR classes into the iOOBN classes, is provided in [31].

Overall, we reduced 129 classes into 8 interfaces, 3 abstract classes and 52 concrete classes, including 36 derived classes. This 50% reduction (129 classes down to 63 components) demonstrates that in large complex models there are indeed common elements that can be more compactly represented by a class hierarchy (rather than as a large number of unrelated classes that actually have many common elements, as in the original WGR OOBN).

This kind of more compact representation also facilitates understanding of the model, and obviously reduces the complexity of the associated documentation. Moreover, extending such a complex model is quite challenging, especially if the original BN modellers and/or the domain experts who build the original version are not longer available. Thus this re-

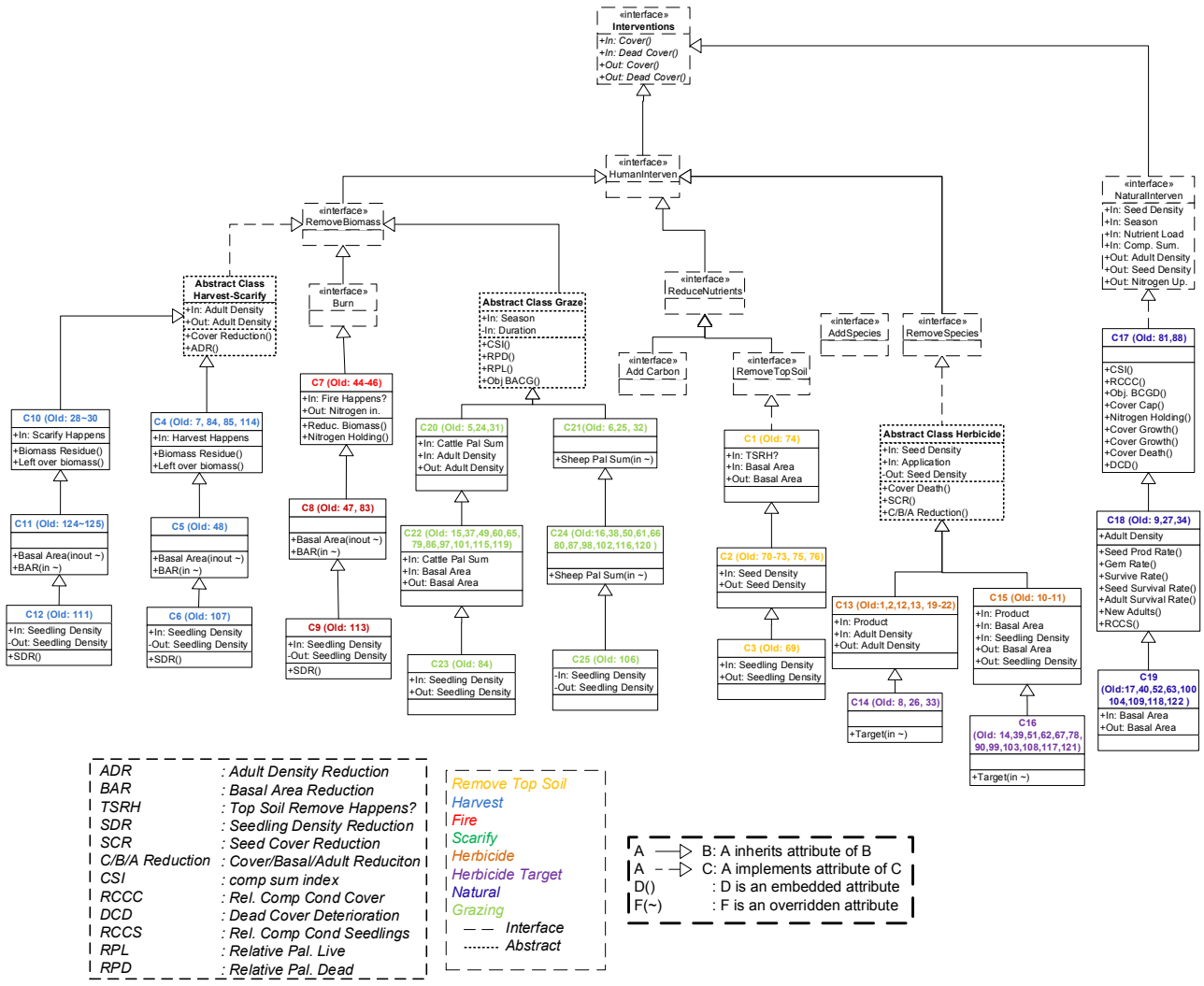


Fig. 14. Class hierarchy (expert opinion and machine learning approach)

engineering case study suggests that the iOOBN framework can be useful for producing more efficient, reusable, extensible and scalable complex BN models.

VI. Conclusions and Future Work

Various forms of object oriented Bayesian networks have been proposed in the literature, to help make BN technology cope with large-scale problems and to support re-use and maintainability. These frameworks has resulted in OOBN systems that contain hierachical composition with encapsulation. However the key OO feature of inheritance, which brings a higher level of resuability and scalability, that has been proposed long ago, have not been formally or fully specified for OOBNs, nor implemented in any OOBN software that we are aware of. In this paper, we have presented an extended iOOBN theoretical framework, which defines inheritance, polymorphism, encapsulation and abstraction for OOBNs. We have implemented a prototype version of the framework using the API of an existing BN software package, and have demonstrated its efficacy by re-engineering an existing large real-world dynamic OOBN into the iOOBN framwork, producing a set of class hierarchies that reduced the number of classes by more than half.

The iOOBN framework has been designed to support team-based development of complex BN models. Its interfaces and abstract classes provide abstraction and should support incomplete and partial implementation of parts of a larger system, where later in the model development process, these segments can be replaced by more detailed or fully concrete classes. The strong type checking within iOOBNs should assist modellers avoid confusion or modelling flaws which may lead to a significant problems in the resultant model. Along with encapsulation and inheritance, type casting and overriding provide scalability and maximization of component reuse.

We are currently using iOOBN to develop a model from scratch for a real-world agriculture domain, which will allow us to evaluate further how it may improve the modelling process and the resultant model efficiency. We are also learning what aspects of the standard BN modelling process need to be adapted for iOOBN modelling. We have already identified that compiling an iOOBN into a flattened BN (as done in the Hugin software) leads to very large complex models that

run into complexity problems for either exact or approximate inference. Thus another planned piece of future research is to adapt the "plug-and-play" incremental compilation algorithm of Bangsø [4] for use with iOOBN, to take advantage of the class hierarchy.

REFERENCES

[1] GeNIe Modeler. https://www.bayesfusion.com/genie-modeler, 2017.

[2] P. A. Aguilera, A. Fernández, R. Fernández, R. Rumí, and A. Salmerón. Bayesian networks in environmental modelling. *Environmental Modelling & Software*, 26(12):1376–1388, 2011.

[3] R.H.A. Baker, A. Battisti, J. Bremmer, M. Kenis, J. Mumford, F. Petter, G. Schrader, S. Bacher, P. DeBarro, P.E. Hulme, O. Karadjova, A.O. Lansink, O. Pruvost, P. Pysek, A. Roques, Y. Baranchikov, and J.-H. Sun. PRATIQUE: a research project to enhance pest risk analysis techniques in the European Union. *EPPO Bulletin*, 39(1):87–93, 2009.

[4] Olav Bangsø, M Flores, and Finn Jensen. Plug and Play Object Oriented Bayesian Networks. *Current Topics in Artificial Intelligence*, 3040(Ic):457–467, 2004.

[5] Olav Bangsø, M. Julia Flores, and Finn Verner Jensen. Plug & Play Object Oriented Bayesian Networks. In *Current Topics in Artificial Intelligence, 10th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2003*, pages 457–467, 2003.

[6] T. Boneh, G.T. Weymouth, P. Newham, R.J. Potts, J. Bally, A.E. Nicholson, and K.B. Korb. Fog forecasting for Melbourne Airport using a Bayesian network. *Weather and Forecasting*, 30(5):1218–1233, 2015.

[7] B. Bruegee and A.H. Dutoit. *Object-Oriented software engineering: Practical software development using UML, Patterns and Java*. Pearson, 3rd edition edition, 2009.

[8] J Cain. Planning Improvements in Natural Resources Management: Guidelines for Using Bayesian Networks to Support the Planning and Management of Development Programmes in the Water Sector and Beyond. Technical report, Centre for Ecology and Hydrology, Crowmarsh Gifford, Wallingford, Oxon, UK, 2001.

[9] T. Charitos, L.C. van der Gaag, S. Visscher, K.A.M. Schurink, and P.J.F. Lucas. A dynamic Bayesian network for diagnosing ventilator-associated pneumonia in ICU patients. *Expert Systems with Applications*, 36(2):1249–258, 2009.

[10] C. Conati, A.S. Gertner, K. VanLehn, and M. Druzdzel. On-Line Student Modeling for Coached Problem Solving Using Bayesian Networks. In *UM97 – Proceedings of the Sixth International Conference on User Modeling*, pages 231–242, 1997.

[11] L. Falzon. Using Bayesian network analysis to support centre of gravity analysis in military planning. *European Journal of Operational Research*, 170(2):629–643, 2006.

[12] Norman Fenton and Martin Neil. Building large-scale Bayesian networks. *The Knowledge Engineering Review*, 15(3):257–284, 2000.

[13] Christophe Gonzales, Lionel Torti, Morgan Chopin, and Pierre-Henri Wuillemin. aGrUM: A GRaphical Universal Modeler. https://forge.lip6.fr/projects/agrum. [Online; accessed 27-June-2017].

[14] David Heckerman, Chris Meek, and Daphne Koller. Probabilistic entity-relationship models, PRMs, and plate models. In *Introduction to statistical relational learning*, pages 201–238, 2007.

[15] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer Verlag, New York, 2nd edition, 2007.

[16] Frank Jensen. Hugin API Reference guide. Technical Report 8, Hugin Experts, 2 2016.

[17] Uffe B Kjaerulff and Anders L Madsen. Bayesian networks and influence diagrams. *Springer Science+ Business Media*, 200:114, 2008.

[18] Daphne Koller. Probabilistic relational models. In *International Conference on Inductive Logic Programming*, pages 3–13. Springer, 1999.

[19] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.

[20] Daphne Koller and Avi Pfeffer. Object-Oriented Bayesian Networks. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, USA, 1997*, pages 302–313, 1997.

[21] Kevin Burt Korb and Ann Elizabeth Nicholson. *Bayesian Artificial Intelligence*. CRC Press, 2010.

[22] K. Kristensen and I.A. Rasmussen. The use of a Bayesian network in the design of a decision support system for growing malting barley without use of pesticides. *Computers and Electronics in Agriculture*, 33(3):197–217, 2002.

[23] Helge Langseth and Olav Bangso. Parameter Learning in Bayesian Networks. *Annals of Mathematics and AI*, pages 221–243, 2001.

[24] Kathryn B. Laskey. MEBN: A language for first-order Bayesian knowledge bases. *Artif. Intell.*, 172(2-3):140–178, 2008.

[25] Kathryn B. Laskey and Suzanne M. Mahoney. Network Fragments: Representing Knowledge for Constructing Probabilistic Models. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, USA, 1997*, pages 334–341, 1997.

[26] Anders L Madsen, Michael Lang, Uffe B Kjærulff, and Frank Jensen. The Hugin Tool for Learning Bayesian Networks. *Learning*, pages 594–605, 2003.

[27] B.G. Marcot, J.D. Steventon, G.D. Sutherland, and R.K. McCann. Guidelines for developing and updating Bayesian belief networks applied to ecological modeling and conservation. *Canadian Journal of Forest Research*, 36(12):3063–3074, 2006.

[28] S. Mascaro, K.B. Korb, and A.E. Nicholson. Anomaly Detection in Vessel Tracks using Bayesian Networks. *International Journal of Approximate Reasoning. Elsevier Science*, 55(1):84–96, 2011.

[29] Christopher Meek and David Heckerman. Structure and Parameter Learning for Causal Independence and Causal Interaction Models. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, USA, 1997*, pages 366–375, 1997.

[30] Judea Pearle. Probabilistic reasoning in intelligent systems, 1988.

[31] Md Samiullah, Thao Xuan-Hoang, David Albrecht, Ann Nicholson, and Kevin Korb. Supplementary materials: IOOBN framework and case study. Technical Report TR 2017/1, Bayesian Intelligence, 2017.

[32] Eran Segal, Dana Pe'er, Aviv Regev, Daphne Koller, and Nir Friedman. Learning Module Networks. In *UAI '03, Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence, Acapulco, Mexico, August 7-10 2003*, pages 525–534, 2003.

[33] A. Tang, A. Nicholson, Y. Jin, and J. Han. Using Bayesian belief networks for change impact analysis in architecture design. *Journal of Systems and Software*, 80(1):127–148, 2007.

[34] Lionel Torti, Pierre-Henri Wuillemin, and Christophe Gonzales. Reinforcing the Object-Oriented aspect of probabilistic relational models. In *European Workshop on Probabilistic Graphical Models*, pages 273–280, 2010.

[35] Owen Woodberry, Jessica Millett-Riley, Ann Nicholson, and Steve Sinclair. An Object-Oriented Dynamic Bayesian Decision Network Model for Grasslands Adaptive Management (Abstract Only). In *the Eleventh UAI Bayesian Modeling Applications Workshop (BMAW 2014), Eds. Kathryn B. Laskey, James Jones and Russell Almond. CEUR Workshop Proceedings, 1218*, 2014.