o3prm Documentation

Release

o3prm team

Feb 16, 2018

Table of contents

The O3PRM language's purpose is to model Probabilistic Relational Models (PRMs) using a strong object oriented syntax.

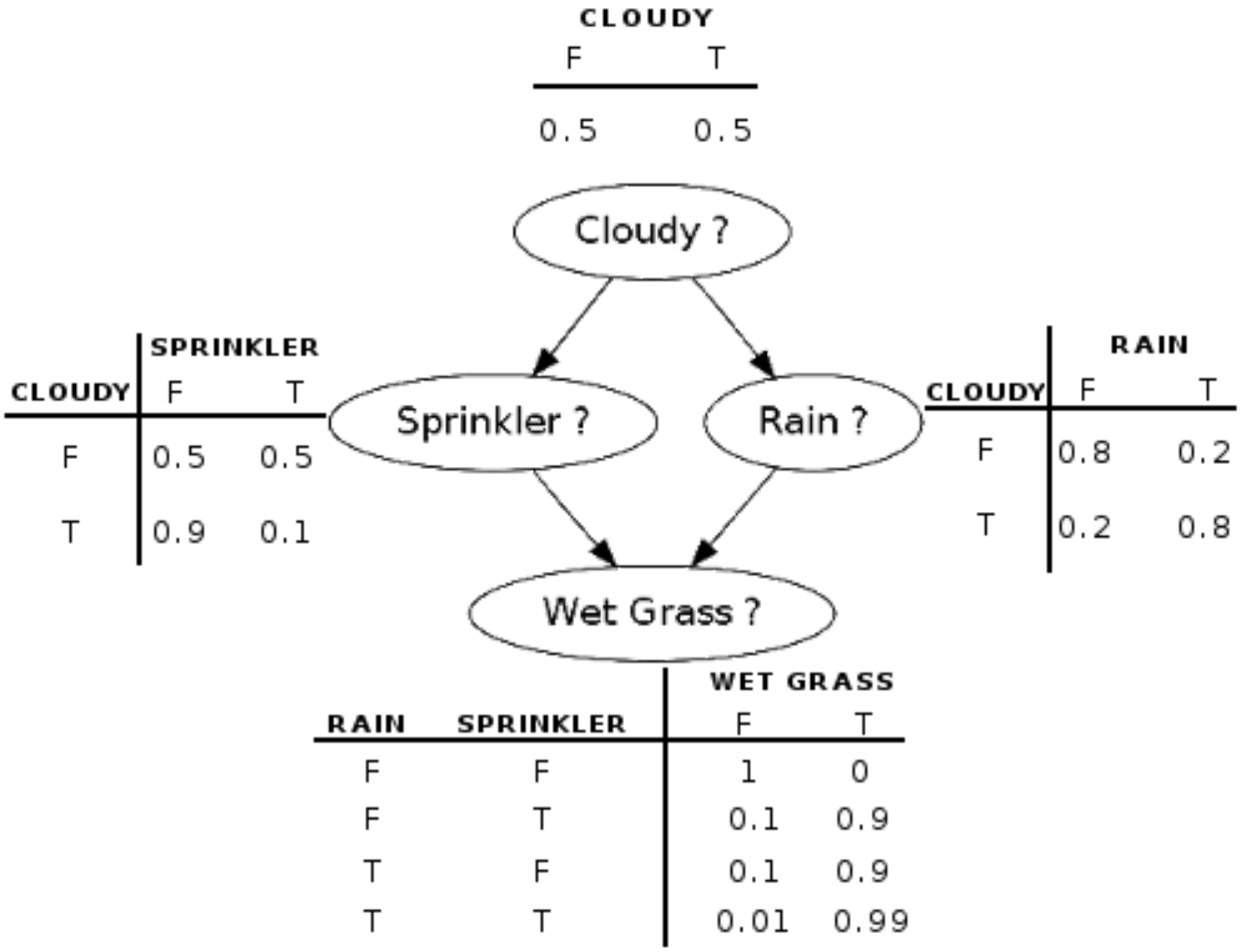
CHAPTER 1

Introduction

Probabilistic Relational Models (PRMs) are a fully object-oriented extension of Bayesian Networks (BNs). Both PRMs and BNs are one of the many Probabilistic Graphical Models. In this introduction, we will offer a very short introduction to both models. For further reading, we suggest the reader to lookup the references in the Bibliography section.

1.1 Bayesian Networks

Bayesian Networks represent discrete probability distributions using a directed graph and parameters called Conditional Probability Tables. The nodes of the graph represent discrete random variables and the arcs represent probabilistic dependencies between those variables (to be precise, the lack of an arc between two nodes represents a conditional independence between the corresponding random variables). To each node is assigned the Conditional Probability Tables (CPTs) of the node given its parents in the graph, and the joint probability distribution over all the nodes/variables in the graph is equal to the product of all the CPTs.



The Water Sprinkler is a classic example of a Bayesian Networks. Its purpose is to infer whether the grass in a garden is wet because of the rain or because the house's owner forgot to turn off the water sprinkler.

The probability distribution is defined over four Boolean random variables: `Cloudy`, `Sprinkler`, `Rain` and `Wet Grass`. There are also four CPTs representing the following conditional distributions: `P(Cloudy)`, `P(Sprinkler|Cloudy)`,`P(Rain|Cloudy)` and `P(Wet Grass | Sprinkler, Rain)`.

We won't go further in our introduction of Bayesian Network and if you never heard of Bayesian Networks, you might need to first familiarize with them before using O3PRM. A good place to start is the pyAgrum tutorials that you can find at http://agrum.gitlab.io.

1.2 Probabilistic Relational Models

Bayesian Networks suffer from the same issue that had early programming languages: the larger the network, the harder it is to create it and to maintain it. Naturally, the BN community looked at the solutions provided by the language programming community and found several paradigms to extend BNs: first order logics, entity-relation models, object oriented paradigms, etc. There is no current consensus on which BN extension is the best as each one offers specific features lacking in the other paradigms.

Probabilistic Relational Models are one of several Object Oriented extensions of Bayesian Networks. They offer a good implementation of the Object Oriented paradigm while not diverging too much from the Bayesian Networks

framework. Their precise definition is provided in subsequent sections of this documentation. But, for the moment, just keep in mind the fact that they are an object-oriented extension of BNs.

1.3 Implementation

The aGrUM framework offers an implementation of the O3PRM language. You can either directly use aGrUM, which is written in C++, its Python wrapper pyAgrum or the prm_run application shipped with aGrUM's source code. All resources and installation instructions can be found at http://agrum.gitlab.io.

CHAPTER 2

Tutorial

2.1 The Water Sprinkler Example

Our first example demonstrates how to represent a Bayesian Network using the O3PRM language. The closest notion of a BN in O3PRM is a class. A class is composed of attributes and relations, we will skip relations for the moment and focus on attributes. Attributes are defined by a set of parents, a CPT and a type.

The attribute's set of parents and CPT are similar to a node's parents and CPT in a BN. However, there is no equivalent of an attribute's type in a BN. Types are used to group random variables with identical domains. The O3PRM language comes with a predefined type for Boolean variables. Boolean Types are declared with the `boolean` keyword and have the domain `false`, `true`. Note that order of the values in the type's domain is important as it determines the meaning of the values in the CPTs.

The following example implements the Water Sprinkler Bayesian Network in the O3PRM language. Each node is represented as an attribute of the `WaterSprinkler` class and all the attributes have the boolean type.

```
class WaterSprinkler {
  boolean cloudy {
    [ 0.5, // false
      0.5] // true
  };

  boolean sprinkler dependson cloudy {
  // false, true => cloudy
    [  0.5,  0.9,  // sprinkler == false
       0.5,  0.1] // sprinkler == true
  };

  boolean rain dependson cloudy {
  // false, true => cloudy
    [  0.8,  0.2,  // rain == false
       0.2,  0.8] // rain == true
  };

  boolean wet_grass dependson rain, sprinkler {
```

```
  //                    wet_grass
  // rain, sprinkler| false, true
        *,          *:  0.1, 0.9;
    false,      false: 1.00, 0.00;
     true,       true: 0.01, 0.99;
  };
}
```

This example shows how to define the set of parents of an attribute using the keyword `dependson`. It also provides two different ways to define an attribute's CPT: either using a tabular declaration (inside square brackets) or a rule-based declaration (see the `wet_grass` CPT).

We strongly recommend formating your CPTs definitions as above to help writing and reading them. See the formatings we used in this tutorial's examples. As you can see, in CPT's tabular declarations, the O3PRM language expects that each column sums to one. In other words, this means that each row of the CPT represents one value of the attribute at the left of the `dependson` keyword. The size of the CPT is the product of the number of rows (i.e., the domain size of the attribute) by the number of columns (i.e., the domain size of the Cartesian product of the attribute's parents).

```
  boolean sprinkler dependson cloudy {
  // false, true => cloudy
    [  0.5,  0.9,  // sprinkler == false
       0.5,  0.1] // sprinkler == true
  };
```

Here, the first value is the probability of `P(sprinkler==false|cloudy==false)`, the second value is `P(sprinkler==false|cloudy==true)`, the third `P(sprinkler==true|cloudy==false)` and the fourth `P(sprinkler==true|cloudy==true)`. You can easily see that each column sums to one.

You can also use rules to declare an attribute's CPT. We recommend to use this syntax when dealing with large CPTs. Each line defines the attribute's probability for a given value of its parents' set. You can also use the wildcard `*` to define the probability for all values of the corresponding parent.

```
  boolean wet_grass dependson rain, sprinkler {
  //                    wet_grass
  // rain, sprinkler| false, true
        *,          *:  0.1, 0.9;
    false,      false: 1.00, 0.00;
     true,       true: 0.01, 0.99;
  };
```

Here, the first line defines the distribution for all possible value of `wet_grass` parents, the following lines overwrite this default distribution defining the probabilities `P(wet_grass|rain==false,sprinkler==false)` and `P(wet_grass|rain==true,sprinkler==true)`. In rule-based declarations, each line must therefore sum to one.

Similarly to any object-oriented progamming language, to use a class, you need to instantiate it, i.e., to create instances of this class. In O3PRM, this is realized in a so-called system. The following shows how to do it.

```
system MyFirstSystem {
  WaterSprinkler water_sprinkler;
}
```

Since we have a single class that defines on its own a probability distribution, we simply need to instantiate it once. But it is possible to create several instances (see the next section) in order to create a world with several gardens and sprinklers.

Finally, we need to define a query using the O3PRM language, to do so you will need to use a different file with the `.o3prmr` extension. With the query language, you can import systems, set observations and query marginal

probabilities of instances attributes.

```
import myFirstPRM;

request MyFirstRequest {
  ? myFirstPRM.MyFirstSystem.water_sprinkler.sprinkler;

  myFirstPRM.MyFirstSystem.water_sprinkler.cloudy = true;

  ? myFirstPRM.MyFirstSystem.water_sprinkler.sprinkler;
}
```

The `import` instruction is mandatory in a request file in order to access the system to query. You simply need to type the name of the file containing the system you want to import, excluding the `.o3prm` extension. You will need to have your `.o3prm` and `.o3prmr` files in the same folder for this to work. You can check Section 2.2 for a better understanding of how import works and how to structure your O3PRM project. For this example, we created the following structure:

Instructions starting with a `?` are queries. They tell the interpreter to compute the marginal probability of the following attribute. To define which attribute to query, you need to write the file's name, system's name, the instance's and finally the attribute's name. Remember that the O3PRM language is case sensitive, event if your operating system is not.

The second instruction assigns an observation (also called evidence) to an attribute. Observations change the beliefs of each node (at least, in most cases). The order among the instructions is important in a request: unlike the first `?` query, the second `?` will take into account the observation and will provide a different marginal distribution for the `sprinkler` attribute. Using the O3PRM interpreter shipped with aGrUM, `prm_run`, you would get the following output:

```
myFirstPRM.MyFirstSystem.water_sprinkler.sprinkler:
        false: 0.7
        true: 0.3
myFirstPRM.MyFirstSystem.water_sprinkler.sprinkler:
        false: 0.9
```

2.2 The Printer Example

In the previous example, we looked at how to model classic Bayesian Networks using the O3PRM language. In this example, we will look into the main features of Probabilistic Relational Models: typing, reference slots and slots chains. These three features, with inheritance, are what differentiate PRMs from BNs and will help modeling large scale probabilistic graphical models. In this example we will not be using inheritance and focus on attribute's type, reference slots, slot chains and aggregators.

A good way to visualize a PRM is to show its class dependency graph.

In this graph, dashed nodes, normal nodes and square nodes represent reference slots, attributes and classes respectively. Arcs either represent probabilistic dependency relations (when solid) or relations (when dashed). We can interpret a class dependency graph as a system where each class is instantiated only once. As a consequence, it can be interpreted as a representation of a Bayesian Network (beware that not all dependency classes represent valid systems because some may involve directed cycles, which is forbidden in Bayes nets since they define incorrect joint probability distributions).

The above example aims to model a simple printer diagnosis problem: we have computers and printers in rooms, each room includes one power supply used to power the equipments it stores. We will start by defining the types used in this example and the `PowerSupply` class.

```
type t_state labels (OK, NOK);

class PowerSupply {
  t_state powState {
    [0.99, // OK
     0.01] // NOK
  };
}
```

The first line declares a categorical type with two possible outcomes: `OK` and `NOK`. The `PowerSupply` class is rather simple: it defines a single attribute `powState` representing the power supply state. The next class introduces reference slots.

```
class Room {
    PowerSupply power;
}
```

Reference slots are class members whose type is another class (it can also be an interface). The key idea behind reference slots is that some attributes may have parents belonging to other classes. Reference slots enable to reach these parents by establishing a link between these other classes and the class of the attribute. The `Room` class only defines a reference slot, pointing towards the `PowerSupply` class. As a result, the attributes in the `PowerSupply` class are accessible in the `Room` class. The next class defines a printer.

```
class Printer {
    Room room;

    boolean hasPaper {
      [ 0.1,  // false
        0.9 ] // true
    };

    boolean hasInk {
      [ 0.3,  // false
        0.7 ] // true
    };

    t_state equipState dependson
      room.power.powState, hasPaper, hasInk { // OK,  NOK
                          *,        *,     *:    0.00, 1.00;
                         OK,     true,  true:    0.80, 0.20;
    };
}
```

The `Printer` class defines a reference slot toward the `Room` class and three attributes: `hasPaper`, `hasInk` and `equipState`. The first two attributes are self explanatory, the third represents the printer's state. The `equipState` attribute has three parents, one of which is the attribute `powState` of the `PowerSupply` class. Since `powState` is not defined in the `Printer` class, we must use a slot chain to tell the O3PRM interpreter where to find it. In this case, the slot chain is composed of two reference slots, `room` of the `Printer` class and `power` of the `Room` class. It ends with attribute `powState` of the `PowerSupply` class.

Finally, the last class defines a computer.

```
class Computer {
    Room room;

    Printer[] printers;
```

```
    boolean exists_printer = exists ( [printers.equipState], OK );

    boolean can_print = and([printers.equipState, exists_printer]);
}
```

The `Computer` class defines two reference slots, `room` and `printers`, and two attributes, `exists_printer` and `can_print`. The `printers` reference slot is different from `room` as its type is suffixed with `[]`. This means that the reference is complex and that more than one printer can be referenced by the `Computer` class. If the number of printers reachable was the same for each computer, we could have created as many instances of class `Printer` as needed and we would not have used keyword `[]`. But what if the number of printers reachable differs from one computer to the other? The keyword `[]` is made for that purpose. It just indicates that there may be zero, one or several printers reachable. When instantiating Class `Computer`, each instance will define its own set of printers. When attributes parents are a slot chain with at least one complex reference slot, the attribute's CPT must be defined using an aggregator (that is designed generically to cope with arbitrary numbers of parents, here of printers). The `exists_printer` attribute illustrates how to declare such attributes using the `exists` aggregator. You can also use aggregators with non complex reference slots, as illustrated with attribute `can_print`.

Aggregators are functions used to generate deterministic CPTs when instantiating classes, i.e., when the exact number of instances referenced by a complex reference slot is known.

2.3 Printers with inheritance

In this example, we will extend the previous printer example with inheritance features of the O3PRM language. Our goal here is to show how you can extend an existing model by using three inheritance tools offered by the O3PRM language: type extensions, class inheritance and interface implementation.

We will first add new types to our model, in order to better represent the semantics of different states each equipment can have.

```
type t_state extends boolean (
  OK: true,
  NOK: false
);

type t_ink extends t_state (
  NotEmpty: OK,
  Empty: NOK
);

type t_paper extends t_state (
  Ready: OK,
  Jammed: NOK,
  Empty: NOK);
```

First we changed the `t_state` type to make it a subtype of the built-in `boolean` type. This will let us use logic functions such as the `and` and `or` aggregators. Type extension syntax is a mapping between the subtype outcomes and the super type ones. Here, we mapped outcome `OK` with `true` and outcome `NOK` with `false`.

We then declared two subtypes of `t_state`: `t_ink` and `t_paper`. Type `t_ink` renames the labels of `t_state` to better represent the semantics of ink cartridges in our example. On the other hand `t_paper` adds a new outcome `Jammed`, mapped to the outcome `NOK`. This helps us distinguish different printer's failure states: the paper tray can be empty or paper can be jammed. Both states prevent from printing but the action to fix the printer's state will differ.

```
class PowerSupply {
  t_state state {
```

```
    ["0.99", // OK
     "0.01"] // NOK
  };
}

class Room {
  PowerSupply power;
}
```

The first two classes are identical with those of the previous example. We now define the `Printer` as an interface instead of a class.

```
interface Printer {
  Room room;
  t_state equipState;
  boolean hasPaper;
  boolean hasInk;
}
```

Interfaces can be viewed as the abstraction of a class: they are defined by a set of attributes and reference slots but they do not define any probabilistic distribution. Classes can implement interfaces, which constrain them to define all the implemented interface's elements. We illustrate this with two new classes: `BWPrinter` and `ColorPrinter` which both implement the `Printer` interface.

```
class BWPrinter implements Printer {
  Room room;

  t_ink hasInk {
    [0.8, // NotEmpty
     0.2] // Empty
  };
  t_paper hasPaper {
    [0.7, // Ready
     0.2, // Jammed
     0.1] // Empty
  };
  t_state equipState dependson room.power.state, hasInk, hasPaper {
    //                      OK,   NOK
    *, *, *:              0.0,   1.0;
    OK, NotEmpty, Ready: 1.0,   0.0;
  };
}

class ColorPrinter implements Printer {
    Room room;
    t_ink black   {
      [0.8, // NotEmpty
       0.2] // Empty
    };
    t_ink magenta {
      [0.8, // NotEmpty
       0.2] // Empty
    };
    t_ink yellow  {
      [0.8, // NotEmpty
       0.2] // Empty
    };
    t_ink cyan {
```

```
      [0.8, // NotEmpty
       0.2] // Empty
    };
    boolean hasInk = forall ( [black, magenta, yellow, cyan], NotEmpty );
    t_paper hasPaper {
      [0.7, // Ready
       0.2, // Jammed
       0.1] // Empty
    };
    t_state equipState dependson room.power.state, hasPaper, hasInk, black {
      //                          OK, NOK
      *, *, *, *:              0.00, 1.00;
      *, *, false, NotEmpty:   0.00, 0.00;
      OK, Ready, true, *:      0.99, 0.01;
    };
}
```

Both `BWPrinter` and `ColorPrinter` define all elements in the `Printer` interface, but with different types. Indeed, in the `Printer` interface attributes `hasPaper` and `hasInk` are both Booleans. In classes `BWPrinter` and `ColorPrinter` they are of type `t_ink` and `t_paper` respectively. This is called type overloading and is legal because both types are subtypes of `t_state`, itself being a subtype of `boolean`. The O3PRM, through the use of cast descendants, ensure that attributes are casted into the proper subtype when used in a CPT.

Finally, class `Computer` has more attributes and illustrates different usages of the `exist` and `and` aggregators. Note that attribute `can_print` casts its parent `equipState` into the boolean type.

CHAPTER 3

O3PRM project structure

The O3PRM language is made of *compilation units* that are placed into *modules*. It is possible to encode in a single file an entire project but it is not recommended. A package matches a specific file in which we we can find at least one compilation unit. The following is a sample O3PRM project:

File extensions must be used as an indicator of the file's compilation unit nature. The following extensions are allowed: `.o3prmr` for queries, `o3prm` for everything else (types, classes, interfaces and systems). It is good practice to name a file with the compilation unit it holds, for example in `computer.o3prm` should contain the definition for the `Computer` class. If the file contains several compilation units, you should name the file according to the unit with the highest order. For example, if you define types, classes and a system in one file, you should use the system's name.

3.1 Compilation units

There exist four different compilation units in the O3PRM language. A compilation unit *declares* a specific element in the modeling process and can either be: an attribute's type, a class, an interface, a system or even a query. Each compilation unit can start with a header. Headers are the locations where you declare `import` statements.

```
<o3prm> ::= [<header>] <compilation_unit> [(<compilation_unit>)]
<header> ::= <import>
<compilation_unit> ::= <type_unit>      |
                       <class_unit>     |
                       <interface_unit> |
                       <system_unit>
```

3.2 Header syntax

Each compilation unit is declared in a module defined by the path from the project's root to the file's name in which it is declared. Directory separators are represented using dots. For example the file `fr/lip6/printers/types.o3prm` defines the namespace `fr.lip6.printers.types`.

Namespaces can be used to import all of the compilation units defined in them, since many compilation units will need units defined in other files. In such cases, we say that a given compilation unit has dependencies which are declared using the `import` keyword. The syntax is:

```
<import>  ::= import <path> ";"
<path>    ::= <word> [ ( "." <word> ) ]
```

A `<word>` is an alphanumerical identifier. You can find its proper definition in the full BNF.

An example:

```
import fr.lip6.printers.computer;
```

The O3PRM interpreter should use an environment variable to know which directory to lookup for resolving compilations units. You should check your O3PRM interpreter to know which environment variable is used.

A compilation unit can be accessed through two different names:

- Its simple name: the name given to it in the file where it is declared (for example `Computer`).
- Its full name: defined by its namespace and its simple name (for example `fr.lip6.printers.Computer`).

In most cases, referring to a compilation unit using its simple name will work, you will need full names only to prevent name collisions/ambiguities. Name collisions happen when two compilation units have the same name but are declared in different namespaces. In such situations, the O3PRM interpreter cannot resolve the name and will raise an interpretation error.

Note that no matter how you refer to a compilation unit (either by its simple name or full name) you must always import it using the package complete name.

Finally, note that compilation units are case sensitive regardless of the operating system.

CHAPTER 4

Type Declaration

The O3PRM language offers three kinds of discrete random variables: categorical (labelized), integer ranged variables and real-valued discretized variables. Since domains can be shared among attributes in a PRM, the random variables' domains should be declared in a separate compilation unit called a `type`.

All types declarations start with the keyword `type` followed by the type's name. The variable's domain is enclosed inside parentheses.

Here is the full entry for types in the O3PRM BNF:

```
<type_unit>       ::= type <word> <type_body>
<type_body>       ::= <basic_type> | <subtype>
<basic_type>      ::= <labelized_type> | <integer_type> | <real_type>
<labelized_type>  ::= labels "(" <word> ( "," <word> )+ )"
<integer_type>    ::= int "(" <integer> "," <integer> ")"
<real_type>       ::= real "(" <float> "," <float> ( "," <float> )+ ")"
<subtype>         ::= extends <path> "(" <word> ":" <word> ( "," <word>)+ ")"
```

4.1 Categorical Types

Categorical types are used to model categorical random variables, such as Booleans or colors (`red`, `green` and `blue` for example). The syntax is straightforward:

```
type t_state labels (OK, NOK);
type t_colors labels (red, green, blue);
```

4.1.1 The boolean type

The O3PRM comes with a single built-in type for Boolean random variables. The type is defined as follows:

```
type boolean labels (false, true);
```

4.2 Integer Types

Integer types are used to model ranges between two integer values. The domain includes all integers between the lower bound and the upper bound specified.

```
type power int (0,9);
```

4.3 Real Types

Real types are used to model discretized continuous variables. There must be at least three values and each interval is defined as `]x, y]`. For example, the following declaration:

```
type angle real (0, 90, 180);
```

defines the 2-valued discrete random variable defined over `]0;90]` and `]90;180]`.

CHAPTER 5

Class Declaration

Classes are the placeholder of attributes and references in the O3PRM language. You can see them as fragments of Bayesian Networks.

```
<class>          ::= class <word> [ extends <word> ] "{" <class_elt>* "}"
<class_elt>      ::= <reference_slot> | <attribute> | <parameter>
```

Classes contain three different elements: attributes, reference slots and parameters.

5.1 Attributes

Attributes are a generic definition of random variables. They are not random variables: only their instances after instantiating the class are random variables. Attributes are defined by a type, a name, a set of parents and a CPT.

```
<attribute>       ::= <attribute_type> <attribute_name> <attribute_cpt> ";"
<attribute_type>  ::= <anonymous_type> | <word>
<anonymous_type>  ::= <labelized_type> | <integer_type> | <real_type>
<attribute_name>  ::= <word> [ <dependency> ]
<attribute_cpt>   ::= ( <CPT> | <aggregator> )
<dependency>      ::= dependson <parent> ( "," <parent> )*

<CPT>       ::= "{" ( <raw_CPT> | <rule_CPT> ) "}"
<raw_CPT>   ::= "[" <cpt_cell> ( "," <cpt_cell> )+ "]"
<rule_CPT>  ::= ( <word> ("," <word>)* ":" <cpt_cell> ";" )+
<cpt_cell>  ::= <float> | """ <formula> """
<formula>   ::= <real> | <function> | <formula> <operator> <formula>
<function>  ::= <function_name> "(" <formula> ")"
<function_name> ::= exp | log | ln | pow | sqrt
```

5.1.1 Tabular Declaration

When declaring a CPT in tabular form, the probability values for all the possible values of the attribute and its parents must be specified. In addition, the order in which the values are specified is important. The O3PRM language uses a declaration by column, i.e., in each *column* of the CPT, the values must sum to one because the rows of the CPT correspond the domain of the attribute for which the CPT is specified (please, note that the terms *column* and *row* are used loosely since the table is only one-dimensional). The following example illustrates the reason why we say that the *columns* sum to one:

```
class Example {
  boolean Y {[0.2, 0.8]};
  boolean Z {[0.5, 0.5]};
  boolean X dependson Y, Z {[
  //      Y==false       |         Y==true
  // Z==false | Z=true | Z==false | Z==true
       0.2,      0.3,        0.7,      0.9,     // X == false
       0.8,      0.7,        0.3,      0.1      // X == true
  ]};
}
```

In this example, we can see that the first value is the probability `P(X=false|Y=false,Z=false)`, the second value `P(X=false|Y=false,Z=true)`, the third `P(X=false|Y=true,Z=false)` and so on.

5.1.2 Rule-based CPT declaration

Rule-based declarations exploit the `*` wildcard symbol to reduce the number of parameters needed to specify the CPT.

```
class Example {
  boolean Y {[0.2, 0.8]};
  boolean Z {[0.5, 0.5]};
  boolean X dependson Y, Z {
    // Y,     Z:   X=false, X=true
       *,  false:  1.0,      0.0;
      true, true:  0.01,     0.99;
     false, true:  0.3,      0.7;
};
```

The first line uses the wildcard `*` for `Y`'s outcomes. This defines in one line the set of probabilities `P(X|Y=y, Z=false)` for `y` in `{false,true}`. There is no limit in the number of rules and, when two rules overlap, the last one takes precedence.

5.2 Reference Slots

Reference slots can either be simple (defining a one to one relation, or a unary relation) or complex (defining a one to N relation, or n-ary relation).

```
<reference_slot> ::= [internal] <word> [ "[" "]" ] <word> ";"
```

5.2.1 Simple Reference Slots

Simple reference slots are used to define a one to one relation between two classes. They are used in slot chains to add parents from other classes to an attribute.

```
class SomeClass {
  boolean Y {[0.2, 0.8]};
  boolean Z {[0.5, 0.5]};
}

class AnotherClass {
  SomeClass myClass;
  boolean X dependson myClass.Y, myClass.Z {
    // Y,     Z:   X=false, X=true
       *,  false:  1.0,     0.0;
     true, true:  0.01,    0.99;
    false, true:  0.3,     0.7;
};
```

Class `AnotherClass` defines the reference slot `myClass` of type `SomeClass` and its attribute `X` uses two slot chains, `myClass.Y` and `myClass.Z`, to reference its parents.

Note that if reference cycles are allowed, you must be careful to not create cycles between attributes. Indeed, if there exists a cycle between two attributes, this implies that the CPT of the first one is conditional given the second attribute and the CPT of the second attribute is conditional given the first attribute. As a consequence, it is not possible to define a joint probability distribution using these two CPTs. The problem is exactly the same for regular Bayesian Networks and it explains why directed cycles are forbidden in BNs.

5.2.2 Complex Reference Slots

Complex reference slots are used to define n-ary relations between classes. They can be used in slot chains when declaring aggregators, special attributes described in section 5.

```
class SomeClass {
  boolean Y {[0.2, 0.8]};
  boolean Z {[0.5, 0.5]};
}

class AnotherClass {
  SomeClass[] myClass;
  boolean X = or([myClass.Y, myClass.Z]);
}
```

To declare a complex reference slots we use `[]` as a suffix to the reference slot type. In the above example, we declared an `or` aggregator referencing attributes `Y` and `Z` accessed though the complex reference `myClass`. Since `myClass` is a complex reference slot, we will be able to reference more than instance of `SomeClass`. Since we do not know how many parents there is for attribute `X`, we need to use an aggregator to generate the attribute's CPT when instantiating the class containing the attribute.

5.3 Parameters

Parameters are used to define constants used in the CPT generation. For example, if we define two parameters such as `lambda` and `t`, we will be able to write the following formula in a CPT: `1-exp(-lambda*t)`.

```
class ClassWithParams {
  param real lambda default 0.003;
  param int t default 8760;
  boolean state {
    ["exp(-lambda*t)", "1-exp(-lambda*t)" ]
```

```
  };
}
```

The `default` keyword is mandatory to provide a default value to parameters, since they can be changed when declaring an instance of a class with parameters.

CHAPTER 6

Interface Declaration

Interfaces are abstract classes used to impose constraints on classes. Just like classes, interfaces have attributes and reference slots (but no CPT). Classes that implement interfaces must necessarily contain the attributes and references slots specified in the interfaces. This mechanism is particularly effective to easily define relations between classes as well as multiple inheritance. For instance, interfaces are the key to define dynamic Bayesian networks (2TBN) using PRMs. Actually, a 2TBN contains one Bayesian network fragment for time slice t=0 and another fragment for transitions between time slice t and t+1, for all t's. Using the 2TBN means copy/pasting the first fragment once, followed by (T-1) copy/pastes of the second fragment, hence resulting in the creation of a Bayesian network over time slices 0 to T. In the O3PRM language, we would naturally consider a class `B0` for the first fragment and a class `Bt` for the second one. As `Bt` models the transition between time slices t0 and t1, some attributes of `Bt` should have parents in `B0` (otherwise the transitions never depend on the past, which makes the temporal nature of the dynamic Bayesian network quite useless). But `Bt` also models the transition between time slices t1 and t2. As a result, the parents that were located in `B0` should now be in `Bt`. So, at first sight, this prevents specifying dynamic Bayesian networks using the O3PRM language. Fortunately, interfaces enable this specification. Actually, in `Bt`, for the transition between t0 and t1, it is useless to know the value of the CPT of the parents belonging to `B0`, what is important is just to know which attributes of `B0` are needed as parents in `Bt`. Similarly, for the transition between time slices t1 and t2, the only information needed in `Bt` is to know which attributes of t1 are used as parents in attributes of t2. As a consequence, if these parent attributes are specified in an interface, and if `B0` and `Bt` implement this interface, both `B0` and `Bt` are constrained to include these parent attributes and we just need to specify that, in `Bt`, the parents of the attributes at time t+1 are those contained in the interface. Since the interface is the same for `B0` and `Bt`, when instantiating these classes, the O3PRM interpreter will select appropriately the parents. Here is the syntax to specify an interface:

```
<interface>      ::= interface <word> [ extends <path> ] "{" <interface_elt> "}"
<interface_elt>  ::= <reference_slot> | <abstract_attr>
<reference_slot> ::= [internal] <word> [ "[" "]" ] <word> ";"
<abstract_attr>  ::= <word> <word> ";"
```

Interface attributes are called abstract attributes because they do not have any CPT.

```
interface MyInterface {
  boolean state;
}

class MyClass {
```

```
  MyInterface iface;
  boolean X dependson iface.state {
    //iface.state==false | iface.state==true
    [                0.2,                0.7, // X==false
                     0.8,                0.3] // X==true
  };
}
```

Interfaces can be used as reference slots types and are useful for defining recursive relations (see the dynamic Bayesian network example described above). Note the keyword `implements` in the example below used to indicate that a class *implements* an interface, i.e., that it declares all the latter's attributes and reference slots. Here, `Base` and `Step` correspond to classes `B0` and `Bt` mentioned in the dynamic Bayesian network example respectively. Note that attribute `state` of `Step` depends on the attribute `state` of Interface `Iface`. As a consequence, when instantiating, `previous.state` can be either the `state` attribute of Class `Base` or that of Class `Step`.

```
interface Iface {
  boolean state;
}

class Base implements Iface {
  boolean state {[ 0.2, 0.8 ]};
}

class Step implements Iface {
  Iface previous;
  boolean state dependson previous.state {
    false: 0.9, 0.1; // P(state|previous.state==false)
    true:  0.2, 0.8; // P(state|previous.state==true)
  };
}

System DynamicO3PRM {
  Base base;
  Step step_1;
  step_1.previous = base;
  Step step_2;
  step_2.previous = step_1;
  Step step_3;
  step_3.previous = step_2;
  // ...
}
```

CHAPTER 7

Functions

Functions are used as placeholders for specific CPTs of classes attributes. They replace the CPT declaration by a specific syntax depending on the type of function used. The first type is the set of functions called aggregators. These functions are used to quantify the information stored in multiple reference slots. The second kind contains deterministic functions and the third probabilistic functions. The last two kinds of functions are not part of the O3PRM specification and are implementation specific. All functions share the same syntax:

```
<aggregator> ::= ( "=" | "~" ) <word> "(" <parents>, <args> ")"
<parents>    ::= ( <parent> | "[" <parent> (, <parent> )* "]" )
<args>       ::= <word> ( "," <word> )*
```

The use of = is reserved for deterministic functions and ~ for probabilistic functions. There are only four built-in functions in the O3PRM language that are deterministic functions called aggregators. There are five built-in aggregators in the O3PRM language: `min`, `max`, `exists`, `forall` and `count`. Other deterministic functions such as `median` and `amplitude` are implemented in aGrUM but they can be implemented in different ways, preventing us from adding them to the O3PRM specification.

7.1 Deterministic Functions

The `min` and `max` functions require a single parameter: a list of slot chains pointing to attributes. The attributes must all be of the same type or share some common supertype. If the common type is not an int, then the type's declaration order is used to compute the min and max values.

```
class Die {
  type int (1, 6) result {["1/6", "1/6", "1/6", "1/6", "1/6", "1/6"]};
}

class GameOfDice {
  Die[] dice;
  type int (1, 6) snake_eyes = min( dice.result );
  type int (1, 6) bingo = max( [ dice.result ] );
}
```

If there is only one element in the list of slot chains the `[]` are optional.

The `exists` and `forall` require two parameters: a list of slot chains and a value. As for `min` and `max`, all attributes referenced in the slot chains list must share a common type or supertype. The value must be a valid value of that common supertype. `exists` and `forall` attribute type must always be a `boolean`.

```
class BWPrinter {
  boolean black   { [ 0.2, 0.8 ] };
}

class ColorPrinter {
  boolean magenta { [ 0.8, 0.2 ] };
  boolean cyan    { [ 0.8, 0.2 ] };
  boolean yellow  { [ 0.8, 0.2 ] };
  boolean black   { [ 0.8, 0.2 ] };
}

class PrinterMonitor {
  BWPrinter[] bw;
  ColorPrinter[] color;

  boolean has_magenta = exists ([color.magenta], true);
  boolean has_cyan = exists ([color.cyan], true);
  boolean has_yellow = exists ([color.yellow], true);
  boolean color = forall([color.black, color.magenta, color.cyan, color.yellow],␣
↪true);
  boolean black = exists( [ bw.black, color.black ] };
}
```

The `count` aggregator counts how many times a given outcome occurred. Its type must be of the form `type int (0, N)`, where `N` is a positive integer. The outcome `N` must be interpreted as "the outcome occurred at least `N` times".

```
class Die {
  type int (1, 6) result {["1/6", "1/6", "1/6", "1/6", "1/6", "1/6"]};
}

class GameOfDice {
  Die[] dice;
  type int (0, 4) four_six = count( dice.result, 6 );
}
```

7.2 Probabilistic Functions

Instead of generating CPTs filled with `0` and `1`, like deterministic functions, probabilistic functions return conditional distributions following a specific rule. A classic probabilistic function is the `noisy-or`, which is implemented in aGrUM as shown below:

```
class NoisyOr {
  SomeIface iface;
  SomeIface jface;
  boolean state ~ noisy_or([iface.state, jface.state], [0.2, 0.1], 0.4);
}
```

As for deterministic functions, the first parameter must be a list of parents. For the noisy-or, the next parameter is a list of weights and the third the noise. These functions are not part of the O3PRM specification and you should check your interpreter documentation for their proper syntax.

CHAPTER 8

Inheritance

Inheritance is a key aspect of the O3PRM language. O3PRM offers four different inheritance mechanisms, all with a specific task. Type inheritance allows to create specialization among random variables' domains. Coupled with type casting, it can be used to model complex problems. Class and interface inheritances offer a more traditional inheritance feature. However its implementation in the O3PRM language adds a lot of expressiveness to Probabilistic Relational Models. Finally, interface implementation is how we implemented multiple inheritance.

8.1 Type Inheritance

Subtypes are used to model a `is a` relation between types. They are declared using the `extends` keyword. You can only subtype categorical types.

```
type t_state labels (OK, NOK);
type t_degraded extends t_state ( OK: OK, DYSFONCTION: NOK, DEGRADED: NOK);
```

Here we declared the type `t_degraded` as a subtype of `t_state`. The mapping notation used inside parentheses indicates how to interpret each of `t_degraded` outcomes as a random variable of type `t_state`.

8.2 Interface Inheritance

An interface can extend another one, using the keyword `extends`. By doing so, the sub interface inherits all of its super interface attributes and references.

```
interface SomeIface {}

interface SuperIface {
  SomeIface myRef;
  t_state state;
}

interface SubIface extends SuperIface {
```

```
  // No need to declare myRef and state:
  // They are inherited from SuperIface.
}
```

8.2.1 Reference Overloading

When you declare a sub interface, you can overload inherited reference slots. To do so, the new reference slot type must be a sub class or sub interface of the reference slot type in the super interface.

```
interface SomeIface {}

interface SomeOtherIface extends SomeIface {
  boolean state;
}

interface SuperIface {
  SomeIface myRef;
  t_state state;
}

interface SubIface extends SuperIface {
  // myRef is overloaded with the sub type SomeOtherIface
  SomeOtherIface myRef;
}
```

8.2.2 Attribute Overloading

As for reference overloading, you can overload inherited attributes with a subtype of the attribute types in the super interface.

```
interface SuperIface {
  SomeIface myRef;
  t_state state;
}

interface SubIface extends SuperIface {
  // state is overloaded with t_state subtype t_degraded
  t_degraded state;
}
```

8.3 Class Inheritance

Class inheritance works the same way as inheritance for interfaces with the additional possibility to overload an inherited attribute's CPT.

8.3.1 Attribute CPT Overloading

To overload an inherited attribute's CPT, you simply need to declare an attribute with a compatible type.

```
class SuperClass {
  boolean state { [ 0.5, 0.5 ] };
}

class SubClass {
  boolean state { [ 0.2, 0.8 ] };
}
```

8.4 Multiple Inheritance

Classes can implement interfaces using the keyword `implements`. When a class implements an interface, it must declare all of the interface's attributes and reference slots. If the class implements several interfaces, then it must declare all the attributes and reference slots of all its interfaces.

```
interface MyIface {
  boolean state;
}

interface MyOhterIface {
  MyIface aIface;
  boolean working;
}

class MyClass implements MyIface, MyOtherIface {
  MyIface aIface;
  boolean state {[0.2, 0.8]};
  boolean working dependson state {
    [0.3, 0.6,
     0.7, 0.4]
  };
}
```

Note that, if a class implements a set of interfaces, then all of its subclasses also implement the same set of interfaces.

8.5 Casting and cast descendants

Casting and cast descendants are how the O3PRM language handles attribute type overloading and probabilistic dependencies. Attributes types and CPTs are tightly coupled: the size of a CPT is the product of the domain sizes of its attribute's type and its parents types. The following example will help us illustrate why we need casting and casting descendants:

```
type t_state labels(OK, NOK);
type t_degraded extends t_state(OK: OK, degraded: NOK, NOK: NOK);

interface Pump {
  t_state state;
}

class WaterTank {
  Pump myPump;

  boolean overflow dependson myPump.state {
  // OK | NOK => myPump.state
```

```
  [ 0.99, 0.25, // overflow == false
    0.01, 0.75] // overflow == true
  };
}

// Centrifugal Water Pump
class CWPump implements Pump {
  t_degraded state {
    [ 0.80, // OK
      0.19, // degraded
      0.01] // NOK
  };
}

system MyPumpSystem {
  WaterTank tank;
  CWPump pump;
  tank.myPump = pump;
}
```

In this example, we model a water tank overflow problem. We have an interface describing pumps, a class representing a water tank and an implementation of interface `Pump` for a centrifugal water pump.

If you look at class `WaterTank` you will notice that its attribute `overflow` depends on `Pump` attribute `state`, which is of type `t_state`.

However, in system `MyPumpSystem`, the reference `myPump` of the instance `tank` of Class `WaterTank` is assigned to an instance of class `CWPump`. Since we overloaded the `Pump.state` type by `t_state` subtype `t_degraded`, the CPT definition of attribute `WaterTank.overflow` should be incompatible.

This is not the case here because a cast descendant of attribute `CWPump.state` is automatically added to the class `CWPump`:

```
t_state state dependons (t_degraded)state {
  // OK, degraded, NOK => (t_degraded)state
  [ 1.0,      0.0, 0.0, // OK
    0.0,      1.0, 1.0] // NOK
};
```

This cast descendant is of the expected type and preserves `WaterTank.overflow` CPT's compatibility.

Attributes added automatically are called cast descendants and can be accessed using the casting notion:

```
<parent>  ::= [ "(" <path> ")"] <path>
```

CHAPTER 9

System Declaration

A system is declared as follows:

```
<system>      ::= system <word> "{" <system_elt>* "}"
<system_elt>  ::= <instance> | <assignment>
```

The first `word` is the system's name. A system is composed of instance declarations and assignments. Assignments are used to assign an instance to an instance's reference slot. The following illustrates a system declaration:

```
system name {
  // body
}
```

9.1 Instance declaration

The syntax to declare an instance in a system is:

```
<instance>  ::= <path> [ "[" digit* "]" ] <word> ";"
```

The first `word` is the instance's class name and the second is the instance's name. For example, if we have a class `A` we could declare the following instance:

```
A an_instance;
```

We may want to declare arrays of instances. To do so we need to add `[n]` as a suffix to the instance's type, where `n` is the number of instances that the array should contain. if `n = 0` then we can simply write `[]`.

```
// An empty array of instances
A_class[] a_name;
// A array of 5 instances
A_class[5] another_name;
```

You can also specify values for parameters when instantiating a class (see Section 5.3 on how to parameterize CPTs). The syntax to do so is:

```
<instance>          ::= <path> <word> "(" <parameters> ")" ";"
<parameters>        ::= instanceParameter ("," instanceParameter)*
<instanceParameter> ::= <word>"="(<integer>|<float>)
```

An example:

```
// We declare an instance of A_class where a_param equals 0.001
A_class a_name(a_param=0.001);
```

9.2 Assignment

```
<assignment> ::= <path> += <word> ";" |
                 <path> = <word> ";"
```

It is possible to add instances into an array, using the += operator:

```
// Declaring some instances
A_class x;
A_class y;
A_class z;
// An empty array of instances
A_class[] array;
// Adding instances to array
array += x;
array += y;
array += z;
```

Reference assignment is done using the = operator:

```
class A {
  boolean X {[0.5, 0.5]};
}

class B {
  A myRef;
}

system S {
  // declaring two instances
  A a;
  B b;
  // Assigning b's reference to a
  b.myRef = a;
}
```

In the case of multiple references, we can either use the = to assign a whole array or the += operator to add instances one by one:

```
class A {
  boolean X {[0.5, 0.5]};
}
```

```
class B {
  A myRef[];
}

system S1 {
  // declaring an array of five instances of A.
  A[5] a;
  // declaring an instance of B
  B b;
  // Assigning b's reference to a
  b.myRef = a;
}
// An alternative declaration
system S2 {
  // declaring three instances of A
  A a1;
  A a2;
  A a3;
  // declaring an instance of B
  B b;
  // Assigning b's reference to a
  b.myRef += a1;
  b.myRef += a2;
  b.myRef += a3;
}
```

CHAPTER 10

Query unit declaration

A query unit is defined using the keyword `request`. Its syntax is the following:

```
<query_unit>  ::= request <word> "{" <query_elt>* "}"
<query_elt>   ::= <observation> | <query>
<observation> ::= ( <path> = <word> ) |
                  ( unobserved <path> )
                  ";"
<query> ::= "?" <path> ";"
```

The first `word` is the query's name. In a query unit we can alternate between observations and queries. An observation, also called an *evidence*, allows to specify the value that we observe for a given random variable (e.g., we observe on our thermometer that variable `temperature` is equal to 20 degrees Celsius). Evidence are assigned to their corresponding random variables using the = operator. A query over random variable `X` asks to infer the probability *P(X|e)* where *e* represents the set of all the evidence specified so far in the request unit. This is done using the `?` operator. The `unobserve` keyword can be used to remove evidence inside the request unit.

```
request myQuery {
        // adding evidence
        mySystem.anObject.aVariable = true;
        mySystem.anotherObject.aVariable = 3;
        mySystem.anotherObject.anotherVariable = false;
        // asking to infer some probability value given evidence
        ? mySystem.anObject.anotherVariable;
        // remove evidence over an attribute
        unobserve mySystem.anObject.aVariable;
        ? mySystem.anObject.anotherVariable;
}
```

For instance, in the above example, the first query over random variable `mySystem.anObject.anotherVariable` returns the *posterior* of the variable given the three evidence entered into the system. The second query returns the *posterior* of the same variable given only the last two evidence entered, the first one being invalidated by the `unobserve` instruction.

CHAPTER 11

O3PRM BNF

11.1 O3PRM Language Specification

```
<o3prm> ::= [<header>] <compilation_unit> [(<compilation_unit>)]

<header> ::= <import>
<import> ::= import <path> ";"

<compilation_unit> ::= <type_unit>   |
                       <class_unit>  |
                       <system_unit>

<type_unit>      ::= type <word> <type_body>
<type_body>      ::= <basic_type> | <subtype>
<basic_type>     ::= <labelized_type> | <integer_type> | <real_type>
<labelized_type> ::= labels "(" <word> ( "," <word> )+ )"
<integer_type>   ::= int "(" <integer> "," <integer> ")"
<real_type>      ::= real "(" <float> "," <float> ( "," <float> )+ ")"
<subtype>        ::= extends <path> "(" <word> ":" <word> ( "," <word>)+ ")"

<class_unit> ::= <class> | <interface>
<class>      ::= class <word> [ extends <path> ] "{" <class_elt>* "}"
<class_elt>  ::= <reference_slot> | <attribute> | <parameter>

<interface>      ::= interface <word> [ extends <path> ] "{" <interface_elt> "}"
<interface_elt>  ::= <reference_slot> | <abstract_attr>

<reference_slot> ::= [internal] <word> [ "[" "]" ] <word> ";"

<attribute>      ::= <attribute_type> <attribute_name> <attribute_cpt> ";"
<attribute_type> ::= <anonymous_type> | <word>
<anonymous_type> ::= <labelized_type> | <integer_type> | <real_type>
<attribute_name> ::= <word> [ <dependency> ]
<attribute_cpt>  ::= ( <CPT> | <aggregator> )
```

```
<dependency>      ::= dependson <path> ( "," <path> )*
<abstract_attr>   ::= <word> <word> ";"

<CPT>       ::= "{" ( <raw_CPT> | <rule_CPT> ) "}"
<raw_CPT>   ::= "[" <cpt_cell> ( "," <cpt_cell> )+ "]"
<rule_CPT>  ::= ( <word> ("," <word>)* ":" <cpt_cell> ";" )+
<cpt_cell>  ::= <float> | """ <formula> """
<formula>   ::= <real> | <function> | <formula> <operator> <formula>
<function>  ::= <function_name> "(" <formula> ")"
<function_name> ::= exp | log | ln | pow | sqrt

<parameter>       ::= param ( <int_parameter> | <real_parameter> )
<int_parameter>   ::= "int" <word> default <integer> ";"
<real_parameter>  ::= "real" <word> default <float> ";"

<aggregator> ::= ( "=" | "~" ) <word> "(" <parents>, <args> ")"
<parents>    ::= ( <parent> | "[" <parent> (, <parent> )* "]" )
<args>       ::= <word> ( "," <word> )*

<parent>   ::= [ "(" <path> ")"] <path>

<system>      ::= system <word> "{" <system_elt>* "}"
<system_elt>  ::= <instance> | <assignment>

<instance>           ::= <path> [ "[" digit* "]" ] <word> ";"
<instance>           ::= <path> <word> "(" <parameters> ")" ";"
<parameters>         ::= instanceParameter ("," instanceParameter)*
<instanceParameter>  ::= <word>"="(<integer>|<float>)

<assignment>  ::= <path> += <word> ";" |
                  <path> = <word> ";"

<word>     ::= <letter> (<letter> | <digit>)
<letter>   ::= 'A'..'Z' + 'a'..'z'+ '_'
<integer>  ::= <digit> <digit>*
<float>    ::= <integer> "." <integer>
<digit>    ::= '0'..'9'
<path>     ::= ["(" ")"] <word> [ ( "." <word> ) ]
```

11.2 O3PRM Query Language Specification

```
<O3PRM> ::= [<header>] <compilation_unit> [(<compilation_unit>)]

<header> ::= <import>
<import> ::= import <path> ";"

<compilation_unit> ::= <query_unit>

<query_unit>   ::= request <word> "{" <query_elt>* "}"
<query_elt>    ::= <observation> | <query>
<observation>  ::= ( <path> = <word> ) |
                   ( unobserved <path> )
                   ";"
<query> ::= "?" <path> ";"
```

```
<word>    ::= <letter> (<letter> | <digit>)
<letter>  ::= 'A'..'Z' + 'a'..'z'+ '_'
<integer> ::= <digit> <digit>*
<float>   ::= <integer> "." <integer>
<digit>   ::= '0'..'9'
<path>    ::= <word> [ ( "." <word> ) ]
```

CHAPTER 12

Examples

12.1 The Water Sprinkler

```
class WaterSprinkler {
  boolean cloudy {
    [ 0.5, // false
      0.5] // true
  };

  boolean sprinkler dependson cloudy {
  // false, true => cloudy
    [  0.5,  0.9,  // sprinkler == false
       0.5,  0.1] // sprinkler == true
  };

  boolean rain dependson cloudy {
  // false, true => cloudy
    [  0.8,  0.2,  // rain == false
       0.2,  0.8] // rain == true
  };

  boolean wet_grass dependson rain, sprinkler {
  //                    wet_grass
  // rain, sprinkler| false, true
        *,         *:  0.1, 0.9;
    false,     false: 1.00, 0.00;
     true,      true: 0.01, 0.99;
  };
}

system MyFirstSystem {
  WaterSprinkler water_sprinkler;
}
```

12.2 The Printer Example

```
type t_state labels (OK, NOK);

class PowerSupply {
  t_state powState {
    [0.99, // OK
     0.01] // NOK
  };
}

class Room {
    PowerSupply power;
}

class Printer {
    Room room;

    boolean hasPaper {
      [ 0.1,  // false
        0.9 ] // true
    };

    boolean hasInk {
      [ 0.3,  // false
        0.7 ] // true
    };

    t_state equipState dependson
      room.power.powState, hasPaper, hasInk { // OK,  NOK
                        *,        *,     *:    0.00, 1.00;
                       OK,     true,  true:    0.80, 0.20;
    };
}

class Computer {
    Room room;

    Printer[] printers;

    boolean exists_printer = exists ( [printers.equipState], OK );

    boolean can_print = and([printers.equipState, exists_printer]);
}
```

12.3 Printers with inheritance

```
type t_state extends boolean (
  OK: true,
  NOK: false
);

type t_ink extends t_state (
  NotEmpty: OK,
  Empty: NOK
```

```
);

type t_paper extends t_state (
  Ready: OK,
  Jammed: NOK,
  Empty: NOK);

class PowerSupply {
  t_state state {
    ["0.99", // OK
     "0.01"] // NOK
  };
}

class Room {
  PowerSupply power;
}

interface Printer {
  Room room;
  t_state equipState;
  boolean hasPaper;
  boolean hasInk;
}

class BWPrinter implements Printer {
  Room room;

  t_ink hasInk {
    [0.8, // NotEmpty
     0.2] // Empty
  };
  t_paper hasPaper {
    [0.7, // Ready
     0.2, // Jammed
     0.1] // Empty
  };
  t_state equipState dependson room.power.state, hasInk, hasPaper {
    //                      OK,  NOK
    *, *, *:              0.0,  1.0;
    OK, NotEmpty, Ready: 1.0,  0.0;
  };
}

class ColorPrinter implements Printer {
    Room room;
    t_ink black   {
      [0.8, // NotEmpty
       0.2] // Empty
    };
    t_ink magenta {
      [0.8, // NotEmpty
       0.2] // Empty
    };
    t_ink yellow  {
      [0.8, // NotEmpty
       0.2] // Empty
    };
```

```
    t_ink cyan {
      [0.8, // NotEmpty
       0.2] // Empty
    };
    boolean hasInk = forall ( [black, magenta, yellow, cyan], NotEmpty );
    t_paper hasPaper {
      [0.7, // Ready
       0.2, // Jammed
       0.1] // Empty
    };
    t_state equipState dependson room.power.state, hasPaper, hasInk, black {
      //                          OK, NOK
      *, *, *, *:              0.00, 1.00;
      *, *, false, NotEmpty:   0.00, 0.00;
      OK, Ready, true, *:      0.99, 0.01;
    };
}

class Computer {
    Room room;
    Printer[] printers;
    boolean functional_printer = exists ( printers.equipState, OK );
    boolean degraded_printer = exists ( printers.equipState, Degraded );
    boolean working_printer = exists ( [functional_printer, degraded_printer], true );
    t_state equipState dependson room.power.state {
      //     OK,  NOK
      OK:  0.90, 0.10;
      NOK: 0.00, 1.00;
    };
    boolean can_print = and([working_printer, (boolean)equipState]);
}
```

CHAPTER 13

Bibliography

J. Pearl. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Infer ence. Morgan Kaufman, 1988.

Daphne Koller and Avi Pfeffer. Object-oriented bayesian networks. In Proceedings of the 13th Annual Conference on Uncertainty in AI, pages 302–313, 1997.

Daphne Koller and Avi Pfeffer. Probabilistic frame-based systems. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), pages 580–587, 1998.

Avi Pfeffer. Probabilistic Reasoning for Complex Systems. PhD thesis, Stanford Uni versity, 1999.

Olav Bangsø and Pierre-Henri Wuillemin. Object oriented bayesian networks: A frame work for topdown specification of large bayesian networks and repetitive structures. Technical report, Department of Computer Science, Aalborg University, 2000.

Olav Bangsø and Pierre-Henri Wuillemin. Top-down construction and repetitive struc tures representation in bayesian networks. In Proceedings of the 13th Florida Artifi cial Intelligence Research Society Conference, 2000.

Olav Bangsø. Object Oriented Bayesian Networks. PhD thesis, Aalborg University, March 2004.

Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Ben Taskar. Probabilistic relational models. In L. Getoor and B. Taskar, editors, An Introduction to Statistical Relational Learning. MIT Press, 2007.

D. Koller and N. Friedman. Probabilistic Graphical Models: Principles and Techniques. MIT Press, 2009.

Judea Pearl. Causality. Cambridge University Press, 2009.

Lionel Torti, Pierre-Henri Wuillemin, and Christophe Gonzales. Reinforcing the object oriented aspect of probabilistic relational models. In Teemu Roos Petri Myllymäki and Tommi Jaakkola, editors, Proceedings of the The Fifth European Workshop on Probabilistic Graphical Models. HIIT Publications, 2010.

Lionel Torti. Structured probabilistic inference in object-oriented probabilistic graphical models. PhD Thesis, Université Pierre et Marie Curie, 2012.