O3PRM Language Specification

Lionel Torti
Pierre-Henri Wuillemin
Laboratoire d'Informatique de Paris VI - UPMC

July 16, 2015

The O3PRM language purpose is to model PRMs using a strong object-oriented syntax. This document purpose it not to explain PRMs but to provide a complete specification of the O3PRM language. Tutorials and examples of PRMs can be found on the O3PRM website.

1 O3PRM project structure

As in Java, the O3PRM language is made of *compilation units* that are placed in *packages*. It is possible to encode in a single file an entire project but it is not recommended. A package matches a specific directory and in each file we can find at least one compilation unit. The following is a sample O3PRM project:

```
fr\
  | lip6\
  |      | printers\
  |      | types.o3prm       // types definition
  |      | powersupply.o3prm // class PowerSupply definition
  |      | equipment.o3prm   // class Equipment definition
  |      | room.o3prm        // class Room definition
  |      | computer.o3prm    // class Computer definition
  |      | printer.o3prm     // class Printer definition
  |      | example.o3prm     // system Example definition
  |      | query.o3prmr      // request query definition
```

File extensions can be used as 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 pratice 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 use the most hight order unit to name the file. For example, if you define types, classes and a system in one file, you should use the system's name.

Compilation units

There exists four different sorts of compilation units. 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 a query. Each compilation unit can start with a header. Headers are where you declare imports statements.

```
<O3PRM> ::= [<header>] <compilation_unit> [(<compilation_unit>)]
<header> ::= <import>
<compilation_unit> ::= <type_unit> |
                       <class_unit> |
                       <system_unit> |
                       <query_unit>
```

Header syntax

Each compilation unit is declared in a namespace defined by the path from the project's root to the file's name in wich it is declared. Directory separators are represented using dots. For example the file `fr/lip6/pritners/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 declered using the `import` keyword. The syntax is:

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

An example:

```
import fr.skoob.printers.computer;
```

The `import` instruction is made of a module's name.

The O3PRM interpreter should use an environment variable to know wich directory to lookup for resolving compilations units. It is recommaned that it behaves like the `CLASSPATH` variable used by Java compilators.

Compilation units can be accessed through two names:

- Its simple name, as it is declared (for example `Computer`).
- Its full name, defined by its namespace and its declaration (for exampe `fr.lip6.printers.Computer`).

In most case, refering to a compilation unit using its simple name will work, you will need full names only to prevent name collision. Name collisions happen when two compilation units have the same name but are declared in different namespaces. In such sitatuation the O3PRM interpreter cannot resolve the name and must 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's namespace.

2 Attribute type declaration

The O3PRM language can only use discrete random variables that are either user-defined or one of the three built-in types: `boolean`, `int` and `real`. User-defined types are declared using the keyword `type`:

```
<type_unit> ::= <built_in> | <user_defined>
<user_defined> ::= <basic_type> | <subtype>
<basic_type> ::= type <word> <word> "," ( "," <word> )+ ";"
```

The rule `<basic_type>` defines a random variable's domain, the first `word` is the type's name and the following are the domain label's names. There must be at least two labels. The rule `<subtype>` is explained in the next section. Some examples:

```
boolean exists;
int (0,9) power;
real (0, 90, 180) angle;
type t_state OK, NOK;
```

Subtyping

A subtype can be declared using the `extends` keyword. A subtype declaration syntax is:

```
<subtype> ::= type <word> extends <word> <type_spec>
<type_spec> ::= <word> ":" <word> (<word> ":" <word>)+
```

The first `<word>` is the type's name, the second the name of its supertype and the rule `<type_spec>` defines label specializations: the first `<word>` is the subtype's label and the second `<word>` is the supertype's label. A example of subtype declaration:

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

In this example, `DYSFONCTION` and `DEGRADED` are specializations of the label `NOK` of type `t_state`. When declaring a subtype, it is mandatory that the supertype is visible, i.e., :

- either the supertype was declared in the same file before the subtype;
- or the supertype declaration unit has been imported.

Built-in types

The built-in types are: `boolean`, `int` and `real`. The `boolean` type is used to represent binary random variables taking the values `false` or `true`. Note that the order used is always `false` first and then `true`. The `int` must be used to define random variables over ranges: `int(0,9) power` defines a random variable power over the domain integers from 0 to 9. The `real` type must be used to define random variables discretized over continuous domains. For example, `real(0, 90, 180) angle` defines a random variable with the two values `[0-90[`, `[90,180[`. When defining a random variable with the `real` type there must be at least three parameters. The syntax of built-in types is:

```
<built_in> ::=
  boolean <word> ";" |
  int "(" <digit>* "," <digit>* ")" <word> ";" |
  real "(" <digit>* ( "," <digit>* )+ ")" <word> ";"
```

3 Class and interface declaration

Classes and interfaces are declared as follow:

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

The first `word` is the class name and the second (if any) the class superclass. An example:

```
class A {
  // reference slots and attributes declaration
}

class B extends A {
  // This class extends class A
  // We say that B is a subclass of A
  // And A the superclass of B.
}
```

Reference slot declaration

In the O3PRM language, simple (1 to 1) and complex (1 to N) reference slots are declared differently. Simple reference slots can only refer to a single instance and complex reference are arrays. The syntax for declaring a reference slot is:

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

The first `word` is the reference slot range's name, we do not called it it's *type* since it could be ambiguous with attributes types. If it is complex `[ ]` are added as suffixes to the range's name and the last `word` is the reference slot's name. The following is an example of two reference slots declaration:

```
// Simple reference slot
A refA;
// Complex reference slot
B[] refB;
```

Attribute declaration

Attributes are declared as follows:

```
<attribute> ::= <word> <word> [ dependson <parents> ]
                ( <CPT> | <function> ) ";"
<parents> ::= <path> ( "," <path> )*
<CPT> ::= "{" ( <raw_CPT> | <rule_CPT> ) "}"
```

The first `word` is the attribute's type, the second its name. Dependencies are defined as a list of parents separated by commas. Each parent is defined by a `<path>`, i.e., a list of reference slots ending by an attribute. We will detail CPTs declaration in the next section. Functions will be detailed in section 6. The following is an example of attribute declaration:

```
// An attribute with no parents
a_type a_name {
  cpt_declaration
};
// An attribute with two parents
another_type another_name dependson parent_1, parent_2 {
  cpt_declaration
};
```

Raw CPT declaration

When declaring a raw CPT, all values of the CPT must be given. In such cases, the value's order is paramount. The declaration used in O3PRM is by columns, i.e., each column in the CPT must sum to one. Let us consider the boolean attributes X, Y and Z such that X depends on Y and Z. The first value in X's CPT declaration will be the probability $P(X = false|Y = false, Z = false)$ and the next value is done by increasing the domain of the last attribute by one. In this case, the second value is the probability $P(X = false|Y = false, Z = true)$. When the last attribute reached its last value, we set it back to its first value and increase the previous attribute. For example, the third value of X's CPT would be the probability $P(X = false|Y = true, Z = false)$. The following illustrates how we can use comments to make raw CPT definitions easier to read.

```
boolean X dependson Y, Z {
  // Y=       |     false    |     true     |
  // Z=       | false | true | false | true |
  /* false */ [ 1.0,    0.3,   1.0,    0.01,
  /* true  */   0.0,    0.7,   0.0,    0.99 ]
};
```

The CPT declaration is dependent on the order in which the parents are declared. The syntax of a raw CPT declaration is straightforward:

```
<raw_CPT> ::= "[" <float>+ ( "," <float>+ )+ "]"
```

Rule based CPT declaration

Rule based declarations exploit wildcards to reduce the number of parameters for CPT with redundant values. Its syntax is:

```
<rule_CPT> ::= ( <word> ("," <word>)* ":" <float> ";" )+
```

There is no limit in the number of rules and when two rules overlap the last rule takes precedence. The following is an example of rule based declaration using the previous example:

```
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;
};
```

Parameters

Parameters are declared using the following syntax:

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

Parameters are either real or integers constants. They are used in CPT definitions to define values using formulas. Some examples:

```
// An integer parameter
int t = 3600;
// A real parameter
real lambda default 0.003;
```

A parameter value can be set at instantiation.

Formulas

It is possible to use formulas in place of float numers to define an attribute's CPT. The formula must be enclosed between quotes and its syntax is implementation specific. For example:

```
class A {
  int t = 3600;
  real lambda = 0.003;

  bool state { [ "1 - exp(-lambda*t)", "exp(-lambda*t)" ] };
}
```

Interface's abstract attributes

An abstract attribute in an interface declaration syntax is:

```
<abstract_attr> ::= <word> <word> ";"
```

Where the first `<word>` is the abstract attribute's type and the second its name.

4 System declaration

A system is declared as follow:

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

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

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

Instance declaration

The syntax to declare an instance in a system is:

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

The first `word` is the instance's class and the second its 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 already added in the array. 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. The syntax to do so is:

```
<instance> ::= <word> <word> "(" <parameters> ")" ";"
<parameters> ::= parameter ("," 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);
```

Affectation

```
<affectation> ::= <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 affectation 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;
  // Affecting b's reference to a
  b.myRef = a;
}
```

In the case of multiple references, we can either use the = to affect an array or the += operator to add instance 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;
  // Affecting 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;
  // Affecting b's reference to a
  b.myRef += a1;
  b.myRef += a2;
  b.myRef += a3;
}
```

5 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 observe an attribute with a given value. Evidence are affected using the = operator. A query over attribute `X` asks to infer the probability $P(X|\mathbf{e})$ where $\mathbf{e}$ is evidence over attributes in the system. This is done using the `?` operator. The keyword `unobserve` can be used to remove evidence over an attribute.

```
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;
}
```

6 Functions

Functions in O3PRM are considered as tools to define attributes CPTs. They replace the CPT declaration by a specific syntax depending to which family the function belongs to. There exit three kinds of functions in O3PRM. The first kind contains built-in functions called aggregators. These functions are used to quantify information hold in multiple reference slots. The second sort contains deterministic functions and the third probabilistic functions. The last two sorts of functions are not built-in functions and are implementation specific. We only provide a generic syntax to keep uniformity between different O3PRM implementations. All functions share the same syntax:

```
<function> ::= ( "=" | "~" ) <word> "(" [ <args> ] ")"
<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`.

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 a int, then the type's values order is used to compute the min and max values.

```
class A {
  // Some declarations
  int(0,10) myMax = max([chain_1, chain_2, ...]);
  // Some declarations
  int(0,10) myMin = max(chain_1);
}
```

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 A {
  // Some declarations
  boolean myExists = exists([chain_1, chain_2, ...], a_value);
}
```

The `count` aggregator behavior is dependant on it's type wich must be of the built-in int type. Then the last value of the aggretors type is interpreted as *at least N occurence of "value"*.

```
type int(0, 5) myRange;

class A {
  myRange myCount = exists([chain_1, chain_2, ...], a_value);
}
```

7 O3PRM BNF

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

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

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


<type_unit> ::= <built_in> | <user_defined>
<user_defined> ::= <basic_type> | <subtype>
<basic_type> ::= type <word> <word> "," ( "," <word> )+ ";"
<subtype> ::= type <word> extends <word> <type_spec>
<type_spec> ::= <word> ":" <word> (<word> ":" <word>)+
<built_in> ::=
  boolean <word> ";" |
  int "(" <digit>* "," <digit>* ")" <word> ";" |
  real "(" <digit>* ( "," <digit>* )+ ")" <word> ";"

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

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

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

<attribute> ::= <word> <word> [ dependson <parents> ]
                ( <CPT> | <function> ) ";"
<parents> ::= <path> ( "," <path> )*
<abstract_attr> ::= <word> <word> ";"

<CPT> ::= "{" ( <raw_CPT> | <rule_CPT> ) "}"
<raw_CPT> ::= "[" <float>+ ( "," <float>+ )+ "]"
<rule_CPT> ::= ( <word> ("," <word>)* ":" <float> ";" )+

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

<function> ::= ( "=" | " " ) <word> "(" [ <args> ] ")"
<args ::= <word> ( "," <word> )*
```

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

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

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

<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> ) ]
```