PCore - the puppet core model - W.I.P PUP-1829

The objective is to enable efficient rich data exchange between two communicating parties that can be separated by time and space as well as platform and implementation language.

This document defines **Pcore** - *the puppet core model* that describes other models. In this document we provide an introduction to modeling and serialization, followed by a more detailed specification of the elements of Pcore.

Why is this needed - what is the problem?

When everything is homogenous and all communicating parties in a system are based on the same implementation technology and all changes in the system are performed in orchestration the communicating parties can simply send data to each other with the simplest mechanisms available e.g. built in serialization like Ruby Marshal (a Ruby version specific binary format).

As soon as the homogeneity is broken all kinds of problems occur. What is it you are sending to me? How is it encoded? What does it mean? You sent me a Ruby object of class `Who::First::Watt`, what am I supposed to do with that? You sent me a data-hash, how am I supposed to know what it represents? Is this data value a Hash, or something that I should take as a description of something more complex? You sent me a number, it is too big to fit in the representation in the programming language used at the other side. (The list goes on).

A perfect example of "who's on first".

Why not JSON all the things, and do what we do now?

While JSON (or YAML) is an excellent interchange format it is also text based and with a very limited set of basic data types. As an example JSON Hash constraints means that it cannot represent all puppet hashes. While this

works fine in contexts where it is known that keys are JSON hash key compliant, the JSON hashes can be used directly. (This is the approach taken in the Puppet Catalog). Unfortunately, this also blocks the ability to send richer data. There is simply no way of encoding the data in an unambiguous way.

We can as an example not send binary data as something distinct - instead it is encoded as a String, and the recipient must know if the data is a String or Binary. This becomes a "point to point" solution where the two end points share this knowledge. While this works, any intermediate processing of the same data will also need to have this knowledge. Our only way to signal that we changed something is to change the protocol version - it does not say anything except that it changed. While we also describe our protocols with a JsonSchema, it does not contain the information that certain attributes on certain resources do special things with the JSON data. You just have to know this, or read the documentation.

It used to be even worse.

In our current 4.x implementation we have made some progress with respect to serialization. We used to have implementations of "to_yaml", "to_pson", "to_this", "to_that" and so on in our ruby classes.

Each with its own quirks. These serializations were also implementations of the shared "point to point" knowledge. This meant that a change had to be performed in multiple locations when the protocol changed. This was improved upon by changing the API to "to_data_hash" and "from_data_hash" - that is, that objects to be serialized could represent themselves with a generic hash. While this centralized the serialization it is still "point to point" and for one use case only.

A particular thorny problem is that of *containment*. As an example, some object have a reference to the "environment" in memory. Does that mean the environment is contained? (Naturally no). In order to fix this, the operation "to_data_hash", puts the name of the environment in the data hash instead of trying to turn the entire environment into a huge hash. Again, the other side must know this is a reference and how to resolve it - it only gets a string with the name - e.g. "production". Guess what happens when something does not use the official API and uses one of the "to_xxx" that the various formats (json, yaml, etc.) inject

everywhere? (Yes, huge waste of memory and performance, and in worst case memory leakage).

And what about data types that would benefit from being in binary form if the serialization format is binary. How do you differentiate one String that represent "binary bytes" from one that is an UTF-8 encoded text String? And in a textual format where the binary has to be Base64 encoded. How can you tell them apart? (If you have not figured it out: *you cannot differentiate them* - the value has to be post processed after deserialization, and that post processing layer must know the actual types of things, or it again becomes a "point to point" implied shared knowledge construct).

Thus, it is still quite bad.

The Bigger Picture Problem

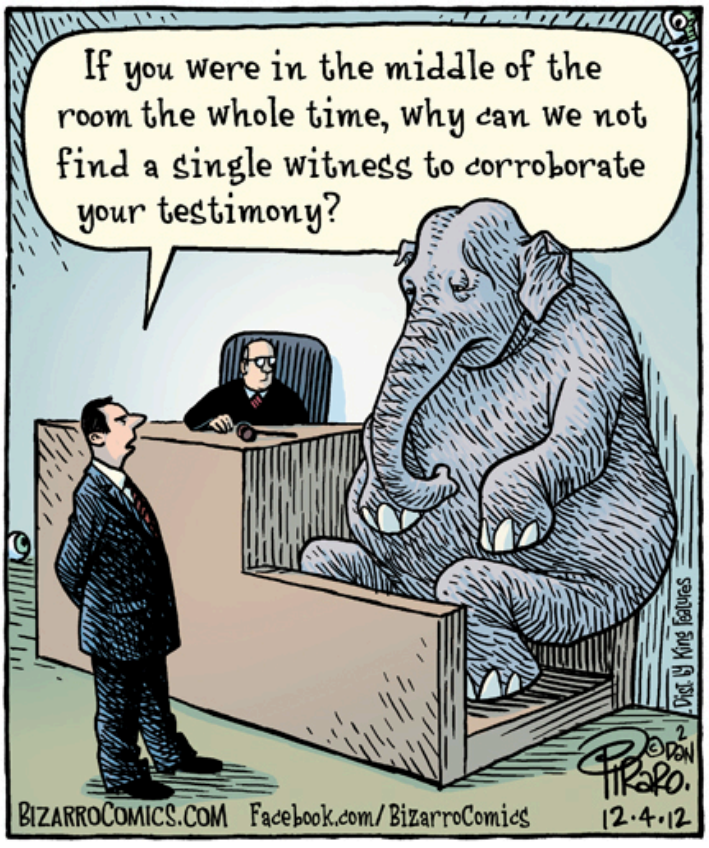
If we take our ear of the wire where the bits are flowing encoded by some serialization technology - what are these bits we are sharing representing if we step back a bit?

"The answer is State. Sometimes a large state."

One party in a group of communicating parties has a state, or if you like a piece of knowledge, that it needs to share with the other members of the communicating group.

Do the communicating parties have the same pre-existing knowledge. Do they see the world the same way, or do they need to be informed about everything every time there is the need to communicate?

Or put a different way; How do we get the Elephant in the room across the wire?



The communication ranges from very simple things "I, the server would like you, the UI, to draw a line from these coordinates to these other coordinates", to "I, the master server wants you, the puppet agent to ensure you have this enterprise application and all of its related dependencies and system level components installed and configured the right way, and keep it that way until I tell you otherwise. Btw, what is your current state?". While the former is not much of an elephant the second certainly is.

While we can solve this problem by not sending you the very same elephant over the wire when nothing has changed we still have to send you the entire elephant even if we just added mascara to a single eyelash.

As everyone that has built a transporter beaming device knows, it is beneficial to cache the transformation - unless Spock's ears have changed, they can be represented by a single photon instead of many. Thus, if "the elephant" can be broken down into individual identifiable parts we only need to send the parts the other party does not already have (or can cheaply compute). Luckily, since we are not sending living complex beings over the wire, but simple abstractions of them the difficulty of the task is greatly reduced :-)

In the domain of Puppet - the kinds of elephants we are pushing around are catalogs describing desired state plus the required logic to process this desired state. The work on direct puppet where file state information is now included in catalogs is a big step in the right direction even if it is a specific "point to point" solution for files in catalogs.

Polyglot - i.e. multiple implementation languages

As we are working on a native implementation of puppet, rich but yet efficient communication becomes fundamental as we need it in additional places compared to today - for example when performing RPC from a native runtime to a co-process that may be written in some other language. This communication needs to be efficient and handle rich data types. Or, to continue with the elephant metaphor - it is not going to be efficient to use an elephant cage to transport microbes.

What Characterizes a Good Solution?

A good solution:

- is as efficient as possible - using hard packed, compressed, binary data
- allows human readable/debuggable representations without changes to domain objects
- allows representation of a model from different viewpoints (compare containment above)
- can handle large and small serializations
- works when parties are in direct contact with each other (negotiates formats etc)
- works when communication is indirect in time/space (for example a file is written today and read a year from now).

- can handle polyglot implementations (for example an agent in C++, a provider in Lua, and a server in Clojure).
- the data is free from serialization concern (for example, data mapped to serialization without having to be designed a certain way).
- the serialized data is free from implementation language concerns (for example, implementation classes in one language are not mentioned).
- formats are free from point-to-point concerns (for example, one serialization of an object does not have to be the same in every serialization).
- shared knowledge has a concrete representation (i.e. more than just a name and a version).
- flexible enough to be efficient when transporting both microbes and elephants, and many of them.

Pcore is a Model you say, what is a model?

Well, not quite *that* kind of model. Although it is kind of the same as what we are talking about. A fashion model is actually an abstraction; research[1] has shown that what we perceive as appealing in people is actually the mean of the characteristics of those around us.

While the model does not look like you or me (at least not me) with our imperfections and quirks - the model is an idealization, a selection of traits that does not include unevenly sized ears, heavy monobrow, and a weak jawline.

This is exactly how we model things in computer programs. We cannot deal with all real world complexities. We have to use an abstraction and a selection of characteristics that are meaningful for what we are trying to achieve with our software project.

Person
name: String
birthdate: Date

We model the things in our domain by drawing boxes and arrows and giving things names.

[1] Read more at: http://facelab.org/

So when we talk about models, we do not mean Marcus Schenkenberg, or Heidi Klum. Our models are boxes with exciting nerdy stuff in them. Both categories are still abstractions though.

Modeling Technology

There are several modeling technologies. One of the oldest (still in use) is UML, which is expressed in what is known as Ecore - the meta meta model for all UML models. If you are now groaning, and thinking: Oh, dear, not UML, then you are absolutely right, but a bit of history cannot hurt. UML defines a plethora of models and diagram types and is quite complex. It is very much associated with enterprise software development in waterfall style projects where it is deemed important to have rigorous blueprints before starting building.

When UML was introduced over 20 years ago they struggled for years with the incompatibilities between different models and the inability to model non object oriented parts of a system (as it meant it was not possible to easily model transformations from the "blueprint" level to a working implementation in a programming language. What they came up with was the Meta Object Facility (MOF). This facility grew in complexity and was eventually simplified at its core into the technology known as Ecore.

In contrast to the rest of the MOF (and UML), Ecore is really useful for everyday development.

From Wikipedia: *"MOF plays exactly the role that EBNF plays for defining programming language grammars. MOF is a Domain Specific Language (DSL) used to define metamodels, just as EBNF is a DSL for defining grammars. Similarly to EBNF, MOF could be defined in MOF."*

MOF is all paper ware. Ecore is a reference implementation (of part of MOF).

We have been using Ecore in Puppet in the form of RGen, an implementation of Ecore in Ruby. (In Puppet 5.0.0 we have replaced RGen with Pcore). We use this primarily for the Puppet AST. We choose this because we knew that we would eventually use a different language platform and why should we do artisan programming in Ruby and end up with code that we could not transform into something else when we could instead model what we needed and see the model as an investment instead of a one-time write off.

On a parallel track to UML and modelling the idea of a *schema* that describes the content of data emerged. First in SGML which became XML, and the derivatives of XML like JSON and YAML. (UML/MOF/Ecore reference for data exchange is XML).

For Json and Yaml there are (now) schemas just like for XML. These technologies are similar to Ecore in that the schemas themselves are expressed as schemas. Thus, having an implementation of the basic technology, and a mapping of the data types it is possible to read/write anything. These technologies are great for interchange but lack the power to be efficiently used to generate runtime representations that are not generic. To do that, a higher order model is needed. As an example, there is nothing in these technologies that can define that something is binary. That modeling level has to take place in a layer on top of what is offered. (A level that is not needed if the "schema schema" (meta meta model) is powerful enough). Thus with these technologies, it becomes the responsibility of the user to appropriately design the high order layer(s) - and this is quite complex. You are also easily fooled into using these technologies in raw form because they are so easy to use. What you end up with is often the "to-amazeballs" discussed earlier, and that there are large chunks of knowledge that needs to be in the heads of the developers of the communicating pieces of software.

For completeness sake, there are many other technologies similar to JSON and YAML and there are many domain specific solutions; Protobuf, and Thrift are two general examples.

What is Pcore - how does it relate to Puppet Language ?

Pcore is based on the Puppet Type system. The Puppet type system is a modern type system well suited for the Puppet Language functional approach using only immutable data. This type system is much richer than just a set of classic data types (for example Integer, String, Array), as it has a set of abstract types such as Variant, Tuple, and Struct and that types are parameterized.

This means that the types in the Puppet Type System can express models with high fidelity without becoming verbose (compared to Ecore and yaml/json-schema, and XML). The Pcore models are thus easier to write, and to read and understand. The fact that the same type system is used in the Puppet Language is a big bonus for Puppet users. (While most also knows JSON and Yaml there is not much use of the json- and yaml-schemas).

Modeling something is now as simple as writing puppet logic - here using the type system with type aliases to define a model (examples shows a simplified part of the Puppet AST).

```
type Expression = Object

type BinaryExpression = Object[{
  parent => Expression,
  attributes => {
    left_operand  => Expression,
    right_operand => Expression,
  }
}]

type ArithmeticOperator = Enum['+', '-', '*', '/']

type ArithmeticExpression = Object[{
  parent => BinaryExpression,
  attributes => {
    operator => ArithmeticOperator,
  }
}]
```

This is not quite Pcore, but very close to it. (Shown this way because a bit more definition is needed before doing the same in Pcore).

Pcore models are expressed in a subset of the Puppet Language

Every Pcore model can be expressed in Puppet Language code. The reverse is however not true; not all Puppet programs are valid Pcore program as Pcore only allows type expressions and literal values.

See "Modeling Concepts" further on in this document for more details.

How does Pcore relate to Ecore ?

Pcore is heavily influenced by Ecore as its inventors managed to get many of the concepts right. There are many things we do not need as Puppet is built on the idea of immutable data. There is simply no need for extensive features that deal with change events and mutation of models when data is immutable. The Puppet Type system is also much richer, as an example Ecore does not even have a Hash/Map type - this must be modelled by hand. As a result everyone cheats and defines a model that is directly tied to a Java implementation. The types in Ecore are parameterized the same way as they are in Java which means that it is not possible to express Cardinality (0:1, 1:1, 0:M, 1:M) directly in the data types, this is instead modeled per attribute in Ecore. In the Puppet type system we can express the same with type parameters and this

makes it both simpler and more compact. Ecore has no Variant type (and this is especially hard to model). (The actual list of small differences is quite long).

As mentioned, we have already been using Ecore in the form of the Ruby gem RGen in Puppet. In Puppet 5.0.0 we have replaced the Ecore/RGen based AST model with one expressed using Pcore.

Puppet Data Types

To start of with something concrete - we have the following *concrete* data types in the Puppet Type System (types marked *TBD* are being designed/implemented as spin-off from this project):

Puppet	**Description**
`Integer`	64 bit signed integer. (No Bignum/Bigint).
`Float`	64 bit IEEE 754 floating point. (No BigDecimal).
`String`	UTF-8 String (0 to 2^32-1 sequence of UTF-8 characters).
`Binary`	A sequence of binary bytes, with an information field describing the encoding.
`Encrypted[T] (TBD)`	Represents an encrypted value of type T. Contains a Binary and has information about the encryption method.
`Array[T]`	0 to 2^32-1 length array of any other data type in defined order
`Hash[K,V]`	0 2^32-1 of (key, value) where key and value may be any other data type. Entries are order based on time of insertion.
`Regular Expression`	A string value that is a regular expression
`Boolean`	false or true
`Undef`	nil, undefined, NULL, etc
`Default`	The symbolic value 'default'
`TypeReference`	A reference to a type like String, Integer[1,10] etc.

`Name / QualifiedName`	A simple name like `apache`, or a qualified name like `apache::port`
`Object (TBD)`	A free form object of a given Type, and SuperType, having a set of named attributes of data types specified by the object's Type.
`ObjectReference (TBD)`	A reference to an Object
`Timespan`	ns duration
`Timestamp`	ns since *Epoch*
`Path (TBD)`	file path (may have subtypes, or be parameterized, absolute, relative, windows, etc)
`URI`	hierarchical and opaque URIs
`Resource`	Base type type for all resource types
`Class`	Base type for all (host) classes
`Callable`	Describes logic that can be called, acceptable parameters etc.
`SemVer`	A SemVer
`SemVerRange`	A range of SemVer versions
`Deferred`	Defers evaluation of a function call to a later time (automatically resolved on an agent when being a value in a catalog).

We also have abstract data types - these are always represented by a concrete type at runtime. When used in a model they help define what is acceptable, or how something should be processed when serializing.

For completeness here are the abstract data types:

Puppet	**Description**
`Tuple[T1, T2,...]`	Like Array, but each entry in the tuple is individually typed
`Set[T] (TBD)`	Like Array but entries must be unique

`Variant[T1, T2, ...]`	An XOR type ("one of").
`NotUndef[T]`	Removes the matching of `undef` if T matches an `undef`
`Undef[T] (TBD, alias for Optional)`	Matches a T and `undef`
`Optional[T]`	Same as Undef[T]
`Struct[ { name => T, ... }]`	Like Hash, but expresses constraints for keys and values.
`Scalar, Number`	
`CatalogEntry`	Base type for resources and classes.
`Init[T]`	A value that can be used to initialize a new instance of type T

Modeling Concepts

This section is a primer in modelling that uses Pcore to illustrate the concepts. The two central concepts *Model*, and *Meta Model* are described in the following meta-levels table:

Meta Level	**Name**	**Synonym**	**Description**
M0	User Objects	Thing	Actual real world objects "the lamborghini parked outside", and conceptual objects such as a file system file, or imaginary objects like a warp engine. In puppet terms: A resource in an operating system, for example the file `'/tmp/foo.txt'`

M1	Model	Data	An abstraction of a User Object (real or conceptual). A data record in memory or on-file e.g. `{type=> 'car', make=>'lamborghini', color=>'blue', reg_nbr => 'ABC123'}`.
			
			In puppet terms: A resource in an operating system that is managed by puppet and described by a resource expression in a catalog, for example:
			`file { '/tmp/foo.txt':`
			`  ensure => present,`
			`  content => 'example text'`
			`}`

M2	Meta Model	Schema	A description of the data/model objects at level M1 described in terms like type, class, interface, attribute, references, features, containment, for example: `type Car = Object[{ make => String, color => RGBValue, reg_nbr => Pattern[/[A-Z]{3}[0-9]{3}/]}]` In puppet terms: expressed in Ruby using the resource type features. Not described in data form.
M3	Meta Meta Model	Meta Schema	A description of the data objects at level M2, described in terms like type, class, interface, attributes, references, features, containment. In puppet terms: Google for Ruby advice and Puppet documentation, read the source of Puppet. Ask a friend.

Examples at Meta Levels 0 to 3

M0 - a real car

M1 - A model of the car in Puppet Language

```
$my_dream_car = Things::Vehicles::Car({
  make    => 'Lamborghini',
  color   => '#fd76c1, # that is, this color
  reg_nbr => 'GOD001',
})
```

M2 - A meta model defining what a Car is in Pcore

```
Pcore::TypeSet({

  name                => "Vehicles",
  namespace_prefix    => "Things::Vehicles",
  name_authority      => "http://example.com",
  pcore_version       => ">= 2016.0.0",
  version             => "1.0.0",
```

```
  types             => {
    RegNbr      => Pattern[/[A-Z]{3}[0-9]{3}/],
    RGBColor    => Variant[Pattern[/#[0-9a-fA-F]{3}/], Integer[0]],
    NamedColor  => Enum[blue, red, pink, black, white, green, yellow]
    Color       => Variant[RGBColor, NamedColor],

    Car => Object[{ attributes => {
      make        => String,
      color       => Color,
      reg_nbr     => RegNbr,
    }
  }],
  # additional things in the Vehicles model
})
```

M3 - A meta meta model defining what a Pcore TypeSet is in Pcore

(This is an illustration of what it looks like, the full definition comes later).

```
Pcore::TypeSet({
  name                => "Pcore"
  name_authority      => "http://puppet.com/urns",
  pcore_version       => ">= 2016",
  version             => "2016.0.0",

  types               => {
    TypeName          => Pattern[/[A-Z]\w*/],
    QualifiedReference => Pattern[/(:?::)?[A-Z]\w*(:?::[A-Z]\w*)*/],

    TypeSet           => Object[{
      attributes => {
        name                => TypeName,
        namespace_prefix    => QualifiedReference,
        name_authority      => URI,
        pcore_version       => SemVerRange,
        version             => SemVer,
    }
  }],
  # The rest of the Pcore types
})
```

What would a M4 meta meta meta model be?

Quite simply: What everything is made of - here appropriately described in the m4 language:

```
define(Fib,‘ifelse(eval($1<=1),1,
$2, ‘ifelse(eval($1==2),1, $3,
‘eval(Fib(decr(decr($1)), $2, $3)
+ Fib(decr($1), $2, $3))’)’)’)
```

Nah, we have no Meta 4 level, we do not need it as we have performed the required short circuiting of what would otherwise be an infinite series of definition-definitions above the Meta 3 level.

You could create a level Meta 4 model though for the purpose of defining different meta meta models - something that unifies the meta meta models of other modeling techniques perhaps.

But what is *it* defined in?

If your inclination at this point is to think about how to model alternative universes and if those are really contained in "all there is" or not, and if the "wizard of oz" is a reality in a parallel universe and that we here now on Earth may be a figment of imagination, then you should probably switch to regular mushrooms.

By Reference or By Containment

We talked about getting the elephant across the wire earlier.

What we really want to do is to leave the elephant behind, and instead hook up with an identical (possibly teleported) elephant on the other side.

We can do this by simply referencing the elephant instead of containing it. In a model this is as easy as this:

```
# Containing an Elephant
MyCircus => Object[{
  attributes => {
    jumbo => Elephant,
  }
}]

# Referencing an Elephant
MyCircus => Object[{
  attributes => {
```

```
    jumbo => ObjectReference[Elephant],
  }
}]
```

From the standpoint of creating an instance of `MyCircus`, it looks exactly the same.

```
$circus = MyCircus({attributes => { elephant => $elephant }})
```

The big question here is "where did the elephant come from?" If this is just a random elephant we happened to find, then we cannot expect the other side to know about it, whereas if it comes from a well defined source then the other side can.

In order to be able to resolve references to objects (the elephant identified as 'Jumbo') there must be a context in which this reference is resolved. In Pcore this context consists of a *Model Set*, and the Model Set contains Models (that may be contained in *Documents*).

Compare this in Puppet where we have a special implementation of this behavior in Direct Puppet where a Catalog is associated with an entire herd of elephants; all the file content that (may) need to cross the wire.

Identity - Who am I, Who are You

T.B.D

- data has no identity $x = 42
- identities in a model set
- identities in a document / model

Model vs. Implementation

Models are only blueprints - you cannot use them directly; they are just description/text (albeit very formalized text). In order to be able to create an instance of for example a Car, set its attributes and write it to disk - or send it off as arguments in a function call we must have something that can be used in a particular programming language to "write code against" so we can create an instance, read/write attributes, build structures etc. This usable representation is what we refer to as "the implementation of a model".

Models that only contain declarative features (name, version, attributes of known data types, etc) can easily be loaded and used without any further action from a user. But a model that contains objects that model operations (i.e. object functions / methods), non declarative

assertions etc. require more work. As the model itself is implementation agnostic; even if it is expressed in the Puppet Language binding of Pcore, there is nothing in this model that describes anything that absolutely requires Puppet or Ruby, or any other implementation language (as long as there exists an implementation of a target platform for Pcore; or even, an implementation that understands just enough of Pcore. While you may equate Pcore at this point with the concrete binding of Puppet Language Pcore to Pcore itself (the subset of the Puppet language used to describe Pcore) - *Pcore is actually just an idea that is represented that way - it is not the only way*. It may be transformed into other forms such as a json schema, it could be represented as a simple implementation in Java Script etc.

A full implementation of Pcore contains (as shown below), both a static and a dynamic implementation and must have a mechanism to combine modeled declarative parts with the target platform implementations of those parts of the model that require logic.

A basic example is implementing the logic for a derived attribute like Person.age which is computed from the current date and the person's birthdate.

In the Puppet Language this could look something like this:

```
type Person implements SomeModel::Person {
  attr age() {
    TimeStamp() - $birth_date
  }
}
```

In the implementation we can include as much logic as we like. We cannot however have that in the model because that would mean that every implementation of Pcore would also have to have a full implementation of the Puppet Language.

Static vs. Dynamic Implementation

A static implementation is one that is expressed in program code; if the model changes the code must be changed. In contrast a dynamic implementation is either generated on the fly, or by representing all objects with a generic implementation.

For some models - like the AST model, it is beneficial to generate a static implementation. There is no need to vary the implementation while the system is running. What we lose in inflexibility; we must generate source code when the model is changed, we gain in start up performance and possibly also in performance when using the objects. Why is that?

A static code generator will simply make all decision out of band with use of the generated code. The generated code only contains exactly what is needed. In contrast a dynamic

implementation will need to dynamically create classes and configure them to become a real useable class. In languages such as Ruby this is very simple as classes are first order objects. It is however more expensive to do so (since the creation of these dynamic classes is performed with logic written in Ruby compared to the Ruby parser which is faster (native or a faster virtual machine than Ruby's). The final alternative is a model driven implementation where one implementation class is used for all kinds of objects, and the implementation uses (lots of) conditional logic (driven by the model).

Thus for anything where we want maximum performance - we want a static implementation, and where we want full flexibility a dynamic implementation. The ability to handle dynamic implementations at the same time as a static implementation is also of importance as it allows for an application to read, understand, and contain logic that operates on a wide range of versions of the implementation. For objects that require implementation (some of its rules/features cannot be described declaratively) there is a choice - maybe it works fine when only reading objects, and maybe it is possible to get the job done without all features being present. A model to model transformation application is a typical example where it would be highly impractical to having to generate code for every model the transformer can read and transform to/from. A dynamic implementation allows models to be loaded and used directly, which is a big win.

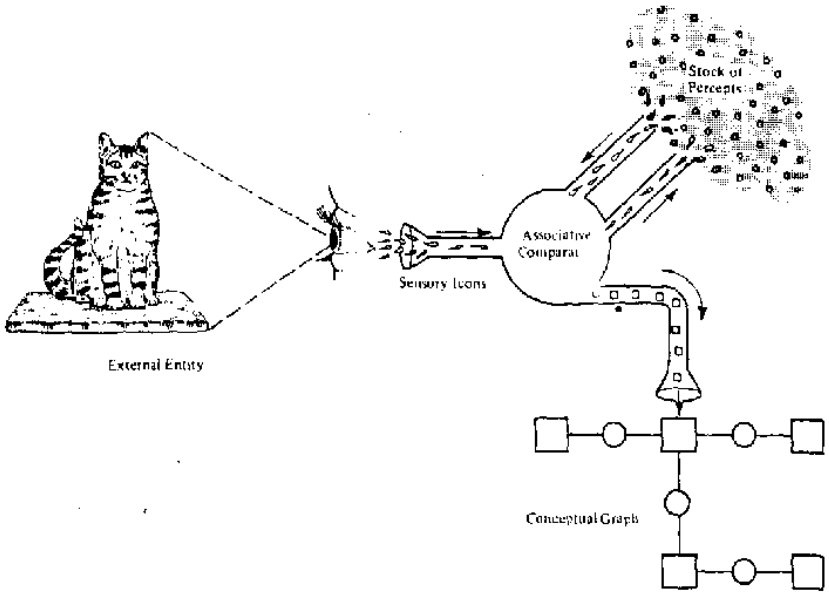
Pcore

Pcore is a way to express a conceptual graph in terms of its own definition. It does this in order to exist.

Its existence must be manifested so we can perceive it.

There are many ways a manifestation can be expressed - ours is in Puppet.

Transformations will make it available in other forms.



TypeSet - what you need to know first

A `TypeSet` is at the root of a Pcore meta-model. (This is called a "Package" in Ecore, but that term was deemed to be to already overloaded in Puppet. You can also think of this as a "Type Module", again a term with strong meaning in Puppet).

As you saw earlier in the discussion about Meta Models and Meta Levels, the definition is recursive; we must have a `TypeSet` to be able to define one. This seems like magic, but there is an implementation of Pcore that backs the definitions - that implementation is the "man behind the mirror" here.

Since `TypeSet` and the `Object` type is what makes Pcore possible we need to start with those. The `Object` is the most fundamental, so it needs to be explained first. `TypeSet` is then defined in terms of an `Object.`

Object Specification

Before looking at `Object` as defined in Pcore, we will use Pcore to illustrate its features.

An Object is similar to a Struct in that it has named features (*keys* in the case of a Struct are akin to *attributes* in the case of an Object). An Object however has other features:

- It can inherit from another Object type
- it can have an API that defines its behavior / operations that can be performed on it
- it has the ability to define what *being equal* is
- and it has the ability to define how objects that are instances of the type are ordered/compared
- In addition, some of the features (for example attributes) have features of their own
- Annotations
- Invariants / Assertions

These features are defined by giving the Object type a hash on this form:

```
Object[{
  parent                => < parent-type >,
  attributes            => { <attributes-hash>  },
  functions             => { <functions-hash>   },
  equality              => [ <equality-array>   ],
  equality_include_type => Boolean,
  order_by              => [ <order-by-tuples>  ],
  assertions            => [ <assertions-array> ],
  annotations           => { <annotations-hash> }
}]
```

All of the entries are optional.

An empty definition such as:

```
Object[{}]
```

is still of value - it is an abstract type that can be used as a marker. Again using the `Vehicle` example - all vehicles may not have any attributes in common, but it is still of value to differentiate them from say Bolivian Fruit Bats, or Foot Salt.

```
if $x =~ Vehicle {
  notice("Let's go for a ride")
}
```

When just using Object without any parameters, then this is a reference to all instances of all types where Object is an ancestor type.

Object Parent

A parent is a reference to another `Object` type. It is illegal to inherit a type that is not a descendant of `Object.` That is, you cannot create a new subtype of `Integer` or `String.` By default if `parent` is not stated, the object type is a direct descendant of `Object.`

The reference is given as a `QualifiedReference` - an uppercase bare word.

```
Object[{
  parent => Things::Vehicle,
  # ...
}]
```

It should be noted right away, that when something is defined in a `TypeSet,` it may use local references to the types in the type set. That is, if the `Object` we just defined is in the same type set as `Things::Vehicle` we would write:

```
Object[{
  parent => Vehicle,
  # ...
}]
```

It is allowed to use fully qualified, but not partially qualified references. That is, if `Vehicle` was defined as `Things::Transportation::Vehicle`, we can use that full name, but not `Transportation::Vehicle.` A local reference has lower precedence than the Puppet Type System - thus if there is the desire to define the type `MyThings::Integer`, it cannot be referenced as just `Integer.`

See `TypeSet` for more details of which types are visible from the Puppet Type System, Pcore, and other type sets.

Object Attributes

The attributes of an Object are always defined as a name and a Type. There are also optional attribute parameters that can be set. The most commonly used attributes are required attributes and there is therefore a short form available for specifying such attributes.

```
Object[{
  attributes => {
    make        => String,
    color       => String,
    chassie_nbr => Integer,
  }
}]
```

When additional parameters are needed the properties are given as a hash. In the short form, the left and right hand side of the `=>` defines the name and type attributes of the attribute, when using a hash, the left hand side defines the name, and the right hand side hash defines all other attributes of the `Attribute`. Here is the Pcore definition:

```
AttributeKind => Undef[
  Enum[ constant, derived, given_or_derived, reference ]
],
Attribute => Object[{
  attributes => {
    name     => UnqualifiedTypeName,
    type     => Type,
    kind     => AttributeKind,
    value    => Any, # constrained by type & kind
    override => {
      type  => Boolean,
      value => default,
    },
    final => {
      type => Boolean,
      value => default,
    }
    annotations => {
```

```
      type  => Undef[Hash[Type[Annotation], Struct]],
      value => undef,
    }
  }
}]
```

`name` attribute

The name must be an unqualified name, (also may not start with '::') as all names are relative to the name of the type set. In the event a qualified name is used it may clash with a referenced type set and its type - that is, if type set A defines Cars::Models as a type, and a referenced model is named Cars and has a type called Models. Such problems are solved by aliasing the referenced model - for example CommonCarTypes is made an alias for the Cars model when it is referenced.

`type` attribute

The type is given as a reference to a type (a `QualifiedReference`). This type must exist earlier in the same type set, in one of the referenced type sets in the `TypeSet` this object is part of, or in the Puppet Type System.

Since the puppet type system has container types; `Array`, `Hash`, `Tuple`, `Struct` it is easy to model complex attributes representing contained values without having to model those as objects. An attribute may also be of `Object` type or one of its subtypes. This means that the object assigned to the attribute is a contained value.

To model that an attribute is a reference (not contained), the type is given as `ObjectReference[T]` where `T` is the type this is a reference to.

```
JournalEntry => Struct[{
  date           => Timestamp,
  driver_license => String[1],
  meter_reading  => Integer
}],

Journal => Array[ JournalEntry ],

Car => Object[[
  log => Journal
```

```
}]
```

If we model that a Car has a reserved parking space, and that a parking space is part of a Garage, we do not want the parking space to be contained in the car. First, it does not make sense, and secondly if we were to send a Car over the wire we would not want to also send the Town containing the Building, containing the Garage containing the ParkingPlace.

```
ParkingSpace => Object[{
  attributes => {
    nbr       => Integer[1],
    leased_by => String
  }
}],
Garage => Object[{
  attributes => {
    parking_spaces => Array[ ParkingSpace ]
  }
}],

Building => Object[{
  attributes => {
    apartments => Array[ Apartment ],
    garages => Array[ Garage ],
  }
}],
Town => Array[ Building ],

Car => Object[{
  attributes => {
    parking_space => ObjectReference[ ParkingSpace ],
    # ...
  }
}]
```

For serialization and resolution see TBD
For how to create a new instance that has attributes that are references see TBD

`kind` attribute

The `kind` attribute defines if the attribute instead of being just *required* or *optional* is `derived`, `constant`, `given_or_derived` or if it is a `reference`.

If `kind` is not specified or set to `undef`, it is required to be given when instantiating unless the `value` for the attribute is set in which case it is assigned that value.

`constant` attribute kind

The `constant` attribute kind defines that the attribute is a constant, its value must be set with `value`.

```
TFord => Object[{
  parent     => Car,
  attributes => {
    color    => { type => Color, kind => constant, value => 'black' },
    # ...
  }
}]
```

A constant value is not serialized, all instances of the same type share this value, it is available because it is defined by the Object Type.

`derived` attribute kind

The `derived` attribute kind controls if the attribute is computed from other attributes.

A value cannot be given for this attribute when instantiating a value of this object type.

The value is not serialized.

The implementation of this object type must have an implementation that computes the value, it is otherwise considered to be an illegal implementation of this type. How this is done is discussed in SECTION TBD.

```
Car => Object[{
  attributes => {
    manufactury_date  => Timestamp,
```

```
    age               => { type => Timespan, kind => derived },
    # ...
  }
}]
```

It is expected that an implementation would lazily produce the derived value and then cache it. The derived value may only depend on the immutable state of the object instance, and other pure functions. If this cannot be guaranteed by a runtime it must compute the derived value at the time the object is instantiated.

`given_or_derived` attribute kind

The `given_or_derived` attribute kind states that if the attribute is given when the object is instantiated this value is used, otherwise it is a computed value. The differences from the `derived` kind are:

- The value is serialized
- It is allowed to give the value when instantiating in which case the given value overrides what would be the computed value.

`reference` attribute kind

A `reference` attribute kind states that a value is referenced but not contained by the object. When the object is serialized the serialization mechanism will serialize a marker that is resolved by a deserializer.

The exact mechanism is TBD.

`value` attribute

The `value` attribute defines a default value for an optional attribute, and the constant value for a constant attribute.

```
Car => Object[{
  attributes => {
    color   => { type => Color, value => 'black },
    # ...
  }
}]
```

`override` attribute

An attribute may override a super type's attribute if it has specified `override => true`. It is an error to specify an override if there is no inherited attribute with the given name.

When overriding, all properties of the attribute must be specified.

- The given type must be the same or a narrower type than the inherited attribute's type.
- It is allowed to override a non constant attribute (implied required or optional, `derived`, `derived_or_given`) with a `constant`.
- It is not allowed to override an attribute of `constant` kind as that would break the contract.

`final` attribute

An attribute may define that it is illegal to override a super type's attribute by stating `final => true`.

Attribute Invariants

(In summary)

- (implied) required attributes may not specify value
- a `constant` attribute must have the value attribute set
- a `constant` attribute may not be overridden
- the same name as an inherited feature may not be used unless `override => true`.
- a `derived`, or `given_or_derived` attribute cannot also specify a `value`.
- it is an error to specify the same attribute property more than once
- It is an error to attempt an override of an attribute that has declared `final => true`

Attributes and functions are in the same name space. When defining an attribute with the same name as an inherited function, this is the same as redefining an inherited derived attribute. The attribute type must be an equal or narrower type than the function's return type.

An attribute cannot redefine a function in the same object type.

This is further detailed in "Object Invariants".

Object Functions

The behavioural aspects of an Object are modelled using functions. This is ideal for describing an API in abstract terms (it becomes concrete when it is bound to an implementation). For

example constraints on a plugin can be described, a service API can be described. A binding to an implementation makes it concrete on a particular platform.

The functions of an Object are always defined as a name and a Callable Type. A function may, just like attributes, have annotations, and it is possible to override a function, but there are no additional attributes that can be set for a function. (One may think that being 'private' would be a feature, but an Object type defines the public characteristics of instances of this type, something that is 'private' is a pure implementation concern).

Callable type is defined in the Puppet Type system and is not described in detail here.

```
Object[{
  functions => {
    name => Callable[...]
  }
}]
```

As a general note, only those functions that are intrinsic to the defined object should be modelled. At runtime there will be many methods added that are meaningful when using runtime objects of this type. As an example, all Ruby objects have the operation 'inspect' and 'to_s', such methods should not be modelled.

Object Function in Pcore

```
Function => Object[{
  attributes => {
    name     => UnqualifiedTypeName,
    type     => Callable,
    override => {
      type  => Boolean,
      value => false,
    },
    final => {
      type => Boolean,
      value => default,
    }
  }
  annotations => {
    type  => Undef[Hash[Type[Annotation], Struct]],
```

```
    value => undef,
  }
}]
```

`name` attribute

The name attribute is the name of the function without namespace. The name must be unique among the types attributes and functions as well as inherited attributes and functions.

It is an error to use a name of something that is defined in the same object type, or in an ancestor type. It is allowed to override an inherited function by specifying `override => true`.

`annotation` attribute

There are no annotations defined for functions in Pcore itself. Use cases for functions include associating information for a code generator on some platform, inclusion of the source code in some scripting language, a default implementation in puppet, etc.

`type` attribute

The type must be a `Callable` that defines the parameters and return type of this function.

`override` attribute

The override attribute allows a function to override an inherited function. It is not allowed to override an attribute. When overriding, the overriding Callable must have compatible signature but may narrow the return type.

`final` attribute

An function may define that it is illegal to override a super type's function by stating `final => true`.

Function Invariants

Functions and attributes are in the same name space. When defining a function with the same name as an inherited attribute, this is the same as redefining that inherited attribute to be derived. The return type of the function must be an equal or narrower type. When doing such an override the default and derived features of the attribute are erased.

Need to describe how object functions can be called in puppet language.
Need to describe how an implementation of an Object can be made in puppet language.

Object Equality

There are four kinds of Equality,

- The very same representation (in memory, often called the *identity*, *object_id*, or similar).
- *Representational Equality* (the same car)
- *Compared Equality* (the same size/value/measure)
- Equal in sort order

Programming Languages typically have different operators for these. Puppet so far has only had the operators `==`, and `!=` for all kinds of equality. We will add the compare operator `<=>` which together with `< <= > =>` are the operators that operate on compared equality.

Instance equality is probably not a concern in the Puppet Language, if required it can be added as a function called `identical(x,y)`.

The representational equality of two objects is by default based on the equality of all of its regular attributes. If an Object type does not have any attributes, it is still different from instances of ancestor and descendant types.

Using the Car from other examples:

```
$car1 = Car({regnbr => 'ABC123', color => 'black'})
$car2 = Car({regnbr => 'ABC123', color => 'black'})

$car_owner_map = { $car1 => mine, $car2 => yours}
```

Would give us an error since we have the same key twice in the same hash. But if we change the example like this:

```
$car1 = Car({regnbr => 'ABC123', color => 'black'})
$car2 = Car({regnbr => 'ABC123', color => 'pink'})

$car_owner_map = { $car1 => mine, $car2 => yours}
```

There is no error since the two instances are not equal because color is different.

Let's change that so that the car's registration number becomes *its representational equality*[2]. We are obviously dealing with a simple view of a Car since the registration number can change over time, and registration numbers are assigned by different authorities in different parts of the world. In our example domain where Car is involved we are not going to care about the color of the car when dealing with example things like parking permits. "Sorry sir, you have a permit to park your pink car with registration ABC123, this one is green - I am going to give you a fine".

We can fix that by specifying which of the attributes to use when determining if two objects are equal.

```
# In the type set
#
Car => Object[{
  attributes => {
    regnbr => RegNbr,
  },
  equality => [ regnbr ]
}]

# and then used like this
#
$car1 = Car({regnbr => 'ABC123', color => 'black'})
$car2 = Car({regnbr => 'ABC123', color => 'pink'})

# This is an error since $car1 == $car2 is true
$car_owner_map = { $car1 => mine, $car2 => yours}
```

Now, if we want to model that different authorities can register cars; there could be a Kazakhstan registered ABC123, and a Kuala Lumpur registered ABC123, and only Borat's Kazakhstan registered ABC123 has a parking permit.

For that we base the representational equality on two attributes:.

```
Object[{
  attributes => {
    regnbr  => RegNbr,
    regauth => ObjectReference[ VehicleRegistrator ],
```

[2] Not to be confused with the identity of the instance (in memory object) that contains the information about a car - that is something different. This is the "representational equality" - two representations of something describe the same thing.

```
  },
  equality => [ regauth, regnbr ]
}]
```

Attribute kinds allowed in Equality

It is not allowed to include constant attributes in equality definitions (they are always the same so it is meaningless to involve a constant).

All other attribute kinds may be used.

Equality in Pcore

```
Object => Object[{
  attributes => {
    # ...
    equality => {
      type  => Undef[Variant[Name, Array[Name]]],
      value => undef,
    },
    equality_include_type => {
      type => Boolean,
      value => true,
    },
  }
}]
```

Equality not given

When equality is not given, the default is equality of all non constant attributes and `equality_include_type` is `true`.

Equality given as a Name or Array of names

When equality is given as a single Name or array of names the `equality_include_type` must be true. Equality check is then limited to the given set of attributes. It is legal to specify an empty list of attributes which means that all instances of the subtype will be equal, but not equal to instances of a super type unless `equality_include_type` is false.

When inheriting an equality definition, the specification of the subtype's equality definition is concatenated to the aggregated ancestor definition.

It is an error to include an attribute more than once in the same equality definition. It is allowed to include an inherited attribute, but only if it is not already part of an ancestor's equality definition.

It is not possible to modify the definition of a super type's equality, only extend it.

Equality that ignores type

In order to ignore type `equality_include_type` should be set to `false`. When type is not included it is illegal to also specify a list of attributes (since a super type does not have those attributes it would be meaningless to specify that type should not be included).

Since it is of value to be able to ignore type in representational equality - as an example, a `Car` type, and a `CarWithFullSpecification` type; instances of these may represent the same real world car, only one with more information. Since this means that no attributes from the subtype can participate, the equality is based only on the type's ancestor attributes.

```
CarWithFullSpecification => Object[{
  parent => Car,
  attributes => {
    data => Hash[String, Any]
  },
  equality_include_type => false,
}]
```

If two siblings of a super type both have `equality_include_type` set to false, then they would compare equal based on their common ancestry.

Object Ordering

This is being discussed in a separate document and will not be part of the first implementation of Pcore.

https://docs.google.com/document/d/1vrU9d6XXaerR4La_gP8gQG1CgvegSQmamue_b-9RLFE/edit#heading=h.i2ztmyyzixns

Object Invariants - Assertions

Assertions allow specification of illegal combinations (invariants). Each illegal combination is specified with an instance of `Assertion`. An Assertion has a message which is the error message issues if the condition described by the assertion is true.

Assertions also supports warnings and deprecations.

Assertions are entered in a list like this:

```
assertions => [
  {
    message => "One way road to the left, sorry",
    when => { direction => 'left' },
  },
  {
    message => "One way road to the right, sorry",
    when => { direction => 'right' },
  }
]
```

The `message` text is used when issuing an error or warning. The `when` part describes a match against the object's state that defines if this assertion should trigger or not. The example above only uses this simple mechanism, additional assertions can be made.
Behind the scenes each entry results in an `Assertion` object without the author having to use the longer form syntax of `Assertion({ })` around each.

Assertion Attributes

- `message` - the message to display when an assertion triggers
- `when` - gates more detailed assertions on a condition (optional)
- `severity` - error, warning or deprecation (default error)
- `defined_is`, `undefined_is` - to control what "being defined" is (optional)
- declarative assertions of different kind and one that allow assertion to be dealt with by the implementation.

message

The `message` is a string that is used as an error or warning message. It may contain interpolation of attributes using variable names in the puppet language long form interpolation syntax - for example

```
message => "Specifying cab_type '${cab_type}' requires model to be
'cabriolet', but it was '${model}'"
```

when

The `when` attribute defines when the Assertion is enabled. It defaults to boolean `true`. It is typically used to gate the assertion on a match of the state of the object (to not waste performance asserting something that is meaningless). It may also be set to `false`, to disable an assertion without having to remove it completely from a model (useful when developing).

The meaning depends on the value of the `when`

value	**enables the Assertion when**
`false`	never enabled
`true`	always enabled
single attribute name	value of attribute is *defined* as determined by the attributes `defined_is` and `undefined_is`.
array of attribute names	when the values of all listed attributes are defined (same way as for single attribute name).
hash of attribute name => match	when all mapped attributes match the respective value (using case option matching semantics; literals, types, regexps etc.). Also see assert_attributes_match.

Allowing XPath expressions instead of just attribute references (which is a very small subset of XPath) alters the semantics somewhat. XPath is powerful and can result in several different kind of derived data type - typically a "sequence"

The array of attribute names is not really needed as that is expressed with a single XPath expression - it allowed it would simply be an alternative way of writing a sequence of XPath expressions -

All the various kinds of assertions can also be achieved with XPath expressions by adding XPath version 3 functions (distinct, union, etc.). it is however not quite the same as matching "at the end" against a Puppet Type system type, or matching against a value.

When validation of an object takes place, all assertions are visited, and if the assertion's `when` matches `true` - the individual assertions are evaluated. If one fails the assertion fails. It is up to the implementation of Pcore in a particular target to define if it reports all failures or stops at the first, and if it fails the creation of the object, or if it requires a validation post creation.

Here is an example where an assertion of a set of attributes are gated on the value 'cabriolet' in the model attribute (in a CarOrder).

```
message => "Cabriolet order inconsistent, see 'Ordering a Cabriolet' in
documentation",
when => { model => 'cabriolet' },
assert_attributes_match => {
  sunroof                          => false,
  ceiling_lining                   => Undef,
  ceiling_mounted_rear_view_mirror => false,
  cab_type                         => NotUndef
}
```

defined_is, undefined_is

The attributes `defined_is`, and `undefined_is` (mutually exclusive) can be used to define what "being defined" means - either as a positive match (defined_is), or as a negative match (undefined_is). This applies per Assertion.

If neither is set, all assertions of *definition* is made with `NotUndef`.

Example:
If all empty objects should be considered to be undefined, this can be specified like this:

```
undefined_is => Variant[Undef, String[0,0], Array[0,0], Hash[0,0]]
```

assert_undefined

Accepts an array of attribute names. This assertion checks that *none of the given* attributes is defined as determined by defined_is, undefined_is.

Example:

```
assert_undefined => x
assert_undefined => [a, b, c]
```

assert_defined

Accepts an array of attribute names. This assertion checks that *all of the given* attributes are defined as determined by defined_is, undefined_is.

Example:

```
assert_undefined => x
assert_undefined => [a, b, c]
```

assert_one_defined

Accepts an array of attribute names. This assertion checks that *one, and only one* attribute is defined as determined by defined_is, undefined_is.

Example:

```
assert_one_defined => [a, b, c]
```

Example:
If being defined should mean that arrays, hashes and strings must have at least one element, this can be achieved with:

```
undefined_is => Variant[Undef, String[0,0], Array[0,0], Hash[0,0]]
assert_one_defined => [a, b, c],
```

which means it is allowed to set a to [], b to "", and c to [1].

assert_none_or_one_defined

Accepts an array of attribute names. At most one of them may be defined defined as determined by defined_is, undefined_is, but they can all be undefined.

assert_one_or_more_defined

Accepts an array of attribute names. At least one of them must be defined as determined by defined_is, undefined_is.

assert_unique_keys

Accepts an array of (at least two) attribute names of attributes being of Hash type. The assertion ensure that a key is only present in one of the hashes.

assert_unique_definitions

Accepts an array of attribute names. If they are defined as determined by defined_is, undefined_is they may not have the same value. Values that are undefined are ignored.

Example:
An old API has modeled a multi valued attribute as a series of individually named attributes instead of using an Array or a Set data type).

```
assert_unique_definitions => [ip_addr1, ip_addr2, ip_addr3]
```

Example:
A bad coin design:

```
Coin => Object[{
  attributes => {
    upside   => Enum[Head, Tail],
    downside => Enum[Head, Tail],
  }
}]
```

Clearly, a real world coin has one head and one tail side - only one can be up at any given moment. We can fix this design with this assertion:

```
assertions => [
  {
    message => "upside and downside cannot be the same",
    assert_unique_values => [upside, downside]
  }
]
```

assert_flattened_unique_definitions

Same as assert_tunique_definitions but the content of attributes are:

- converted to an Array if not already being an array
- all attributes are concatenated and flattened
- uniqueness check is made on the result

Example:
A type has two attributes `low_prio`, `high_prio` - the same value may only be listed once in one of them.

```
assert_flattened_unique_definitions => [low_prio, high_prio]
```

assert_attributes_match

Accepts a hash mapping attribute name to a match; i.e a value or a type. The defined_is, undefined_is has no effect on the outcome for this assertion. The matching uses puppet language case option semantics where values are matched against literal values, regexps, the literal default or a Type.

Example - dependent values:
When ordering a 'cabriolet' you cannot get it with a sunroof, or anything related to a fixed 'hard-top'.

```
message => "Cabriolet order inconsistent, see 'Ordering a Cabriolet' in
documentation",
  enabled_when            => { model => 'cabriolet' },
  assert_attributes_match => {
    sunroof                          => false,
    ceiling_lining                   => Undef,
    ceiling_mounted_rear_view_mirror => false,
```

```
    cab_type                          => NotUndef
 }
```

Sidebar: It would be very useful to have a NotMatches object that negates the match. This is otherwise very difficult to achieve for literal values. e.g. a NotMatches(value). This is different than a Not type (which is not wanted because it makes type inference very problematic)

Example - constrained conditional values:

An Order for Cheese indicates the type of cheese, and the thickness of wanted slices. Clearly, you cannot slice mozzarella wafer thin.

```
cheese    => Enum[mozzarella, provolone, swiss],
thickness => Integer[1, 100],
```

```
assertions => [
  {
    message    => "mozzarella slices are only available in thickness 10-50mm",
    when => { cheese => mozzarella },
    assert_attributes_match => { thickness => Integer[10,50] },
  },
]
```

assert_attributes_exist

Asserts that one or more given attributes contain attribute references that exists in the same type, or a given type. This assertion accept a single attribute name, a list of attribute names, or a struct `Struct[{attributes => AttributeNames, type => Undef[Type]}]`.

The assertion logic:

- skips all undefined entries and empty arrays
- if an attribute is a hash, the keys must be of type Name, and these names are added to the names to assert
- If an attribute is a Name, it is added to the names to assert
- If an attribute is an Array[Name] the names are added to the names to assert
- All other data types (except Undef) yields an error (it is not a Name)
- If a type is not given, the 'self' type is the type to assert the names against

- All names to assert are asserted to be an attribute specified in the type

This is best illustrated by the assertion made for the Assertion type itself since it contains several attributes that name attributes in the same type.

```
{ message => "Attribute reference to non existing attribute",
      assert_attributes_exist => [
        assert_undefined,
        assert_one_defined,
        assert_none_or_one_defined,
        assert_one_or_more_defined,
        assert_unique_keys,
        assert_unique_definitions,
        assert_attributes_match,
        assert_flattened_unique_definitions,
        assert_attributes_exist ]
    },
```

An implementation of this assertion must append the name of the attribute holding an illegal attribute reference, as well as the illegal attribute reference, and the type to the message string given in the assertion (e.g. "'<type>.assert_undefined' contains a reference to the attribute '<type>.fooo' which does not exist").

asserted_by

This attribute describes that additional assertion is performed by a referenced function. The referenced function must be a function defined in the same type (it may be inherited). One and the same function may be used in multiple Assertions.

When an assertion of an object is made, all assertions that are enabled (as specified by the `when` attribute resolving to true) are collected. The unique set of all asserted_by functions is then collected, and each function is called exactly once. The results are then remembered as each Assertion that is enabled is evaluated.

The referenced function must be a Callable that accepts an Array of unique assertion codes (one per Assertion that references the function), and that produces an assertion result in the form of a Hash, where a result is bound to each assertion code. The value for an assertion code is Variant[Boolean, String], where a false result means that the assertion failed, a true value that assertion succeeded, and a String result that the assertion failed with a detailed message.

Here is an example:

```
Assertions => Array[Assertion],
AssertionResult => Hash[String => Variant[Boolean, String[1]]],

SomeType => Object[{
  attributes => {
    x => Integer,
    y => Integer,
    z => Integer,
  }
  assertions => [
    { message => "x must be at least 2*y, actual values {x => ${x}, y => ${y}",
      asserted_by => validate_dimensions,
      assertion_code => 'X_LESS_THAN_2_Y'
    }
    { message => "z must be larger that x + y",
      asserted_by => validate_dimensions,
      assertion_code => 'Z_LESS_THAN_X_Y_SUM'
    }

  ]
  function validate_dimensions => Callable[Assertions] >> AssertionResult,
}]

# in the implementation
type SomeTypeImpl implements SomeType {
  function validate_dimensions(Assertions $assertions) {
    $assertions.reduce({}) |$result, $a| {
      case $a[assertion_code] {

        'X_LESS_THAN_2_Y': {
           if $x < 2 * $y {
             $result + { $a[assertion_code] => false }
           } else {
             $result
           }
        }

        'Z_LESS_THAN_X_Y_SUM': {
          if $z < $x * $y {
            $result + { $a[assertion_code] => "${z} < ${$x+$y}" }
          } else {
            $result
          }
        }
        default : { fail("Unknown assertion code: '${$a[assertion_code]}") }
      }
    }
  }
```

```
}
```

The implementation of each assertion can naturally be broken up into a function per validation. While this may provide an implementation that is less complex it may also clutter the object type with a flea market of validation functions and this may be unwanted in the object type interface.

Combining assertions

It is allowed to use multiple assertion kinds in the same Assertion. If one assertion fails, the Assertion fails.

Describing assertions that are only available in an implementation

If we want to describe an implementation as an Object and the implementation has additional assertions, we want to include those assertions in the Object type (the interface), but we cannot include their implementation (puppet code, or whatever it may be implemented in).

The rationale for including these are to define the messages and assertion codes that constitute the API of the object type. (These assertions can be expected).

Exactly how an implementation binds these additional validations to an implementation is unspecified. The implementation may if so possible validate that all assertions that are declared in the object type are also implemented, but this may not be possible (e.g. if all such validations are performed by one single function/method)

An implementation is required to honour the assertion codes and messages, and issue the implementation specific bindings of such object type defined assertions the same way as it supports the built in assertions.

Identity and inheritance of assertions

Sub types inherit all assertions; there is no way they can be avoided without the author of the base type includes an overridable feature that is checked in the base type.

Since messages are not directly used as keys, there may be more than one entry that results in the same message. (Users should be encouraged to use unique messages to make them easier to identify.

I18N / assertion code

To allow translation of assertions, an assertion_code => Optional[String] may be set to an identifier string. This code may be used by tools to correlate validation errors, serve as an index key in translations of messages etc. The use is unspecified. A runtime should include it in warnings/errors if it is present.

Example - constrained conditional values:

```
assertions => [
  {
    code       => 'MOZZARERROR',
    message    => "mozzarella slices are only available in thickness 10-50mm",
    when => { cheese => mozzarella },
    assert_attributes_match => { thickness => Integer[10,50] },
  },
]
```

Severity

By default the severity is error. This can be set to deprecation, warning, or error. (With obvious outcomes).

Assertion Invariants

- An Assertion that only specifies a when is illegal
- Multiple Assertion instances may use the same message
- Multiple Assertion instances may use the same assertion code
- Assertions may use
    - inherited attributes
    - constant attributes
    - derived (or derived_or_given) attributes

Making Assertions more powerful

Since all assertions are basically textual representation of attributes we can make these references far more powerful by adopting XPath syntax. This enables being able to express advanced queries and assertions. Say we want to constrain that all cars in a garage must have the same color.

```
Car => Object[{
  attributes => { color => String }
}],
Garage => Object[{
  attributes => { cars => Array[Car] }
}]
```

We cannot address that with just the Type system (it is not powerful enough yet to extract type parameters from actual values - if it could we could express that the the Array of Cars should be a homogenous set of cars based on Color, and we would then see a type error.

Using an XPath approach, we can instead select the values we want to assert - instead of just selecting attribute values of the current object, we can collect any addressable set of values (and even do filtering).

With an XPath of './cars/color' - we would select the attribute color from each and every car and form an array of the result. We can then match that against a Set[String, 1, true] since if a homogeneous array (all values are the same) was used to create a Set, it would result in a Set of 1. (The Set type is not yet implemented though - here it is invented that the last parameter 'true' indicates that it should create a set out of the LHS before matching).

```
assertions => [
    { message => "All cars must have the same color",
      assert_attributes_match => { './cars/color' => Set[String, 1, 1, true] }
    }
]
```

The Complete XPath has quite a lot of functionality, including comparison operators and arithmetic, functions to pick the first/last etc, and to navigate the entire document module in which an object occurs. To support the full specification there are additional things to solve - like identity/containment and being able to navigate to the container of an object.

With Xpath supported for selection of values - we can directly support advanced constraints that would otherwise require an implementation to express the logic. Having XPath support will make representation of other models in the Puppet domain more easily expressed (as an example, Yang models makes use of XPath in "must" constraint clauses).

If we go down this path [sic] we will need to revise the set of assertions as they may not all make sense - or can be expressed more succinctly with XPath plus a match (e.g. using count

and a select and matching against the value). We may also want to use XPath to express the value(s) to match against instead of just literals and types. With the assert_attribute_match as it now stands this means having to create instances of XPath object as matching values and somehow giving them the 'current node' as the starting point for the expression). Inventing another assertion would be doable where both LHS and RHS are XPath expressions could be one way. XPath in itself can express selection of nodes where the values of some nodes are constrained by values elsewhere expressed by XPath - those expressions become quite difficult to write. At the end there is always the need to compare the result against something - is it empty, size 1, a unique set, etc. which means it may be enough to just have XPath selection where we current have specified an attribute name.

We should explore this more before making the first implementation of assertions in Pcore. We could still implement a first version where only simple attribute names are allowed if that will work in a backwards compatible way with using XPath queries instead of just attribute names in a later version. (e.g. maybe by equating 'name' with './name')

Annotations

Annotations allow information that is not modeled directly by modeled object to be "anonymously contained" by objects in a model. This is used to include additional metadata that is for the consumption of some consumer that knows the meaning of the annotation. This could for example be implementation specific directives that are best maintained at the model source, it could also be information that is use to enable transformations without losing information when a roundtrip is performed.

The mechanism works like the puppet adapters used at runtime. It is required to give the type of an adapter, a reference to what is being adapted and a data structure with the data associated.

Here is the Pcore specification:

```
annotations => {
  type => Undef[Hash[Type[Annotation], Struct],
  default => undef
}
```

Annotations are available both at the object and at the attribute level. Annotations are optional.

To include annotation in a model, an association between an Annotation type, and a Struct defining the attribute values of this annotation is used.

As an example these annotation types are defined:

```
# In a TypeSet named GraphicalMarkup
Position => Object[{
  parent => Annotation,
  attributes => {
    x => Integer,
    y => Integer
  }
}
Icon => Object[{
  parent => Annotation
  attributes => {
    image_url => URI,
  }
}
```

Then, a modelled Car type and instances of a Car can include these annotations directly in a model.

```
$my_dream_car = Pcore.annotate(
  Things::Vehicles::Car({
    make    => 'Lamborghini',
    color   => '#FFC0CB',
    reg_nbr => 'GOD001',
  },
  { GraphicalMarkup::Position => { x => 500, y => 500 }}
)

# In an Object definition in a TypeSet

Car => Object[{
  attributes => {
    make        => String,
    color       => Color,
    reg_nbr     => RegNbr,
  },
  annotations => {
    GraphicalMarkup::Icon => {
      icon_url => http://data.whicdn.com/images/80745698/large.jpg
    }
  }
}],
```

These annotations can be used as a default icon and a default position for that icon on a screen, if this instance of a car was to be rendered on a screen.

Typically annotations are used at runtime. The adapter pattern is then used to read and set the adapter. This is implementation specific. As an example, to modify the position from the default this may be how that is expressed in Ruby.

```
GraphicalMarkup::Position.annotate($my_dream_car) || {
  { x => 0, y => 0 }
}

$my_dream_car = Pcore.annotate(
  Car('Lamborghini', Color('red'), 'ABC123'),
  { GraphicalMarkup::Position => { x => 0, y => 0 },
    InsuranceBracket => { name => 'ridiculously high' }
  })
```

To just get values from an adapter:

```
$position = GraphicalMarkup::Position.annotate($my_dream_car)
$x = $position.x
$y = $position.y
```

This can be used in a use case where a model is read into memory and instantiated, say in JavaScript in a Browser. The model is visualized and elements are dragged around by the user. As their positions change, the respective adapters are updated to hold the position. When the user wants to save this as a document, the entire model can be serialized, and the annotations serialize as contained values with the objects.

Rules for annotations:

- At any given time there is at most one annotation per annotation type associated with an object. (If an adapter needs to hold values for multiple clients, this is up to the adapter implementation).
- An implementation is free to make the object to annotation association in any way that is suitable (it is optional), but if it is supported annotations should be read and written as contained in the object they are associated with. In general an implementation should require access to annotations via each annotation type (Type.adapt(instance))

- When annotations are used in a Pcore model, the annotation types must be defined in a type set that is referenced (or in the same type set).
- At runtime there is no requirement that annotations are referenced in the typeset unless the model is serialized for the purpose of being deserialized by someone else. In that case, a new typeset must be defined that references the data type set (the things being annotated), and the type set(s) for all used annotations. This is prudent to do even if the serialization/deserialization is considered to be in a private domain.

Complete Object in Pcore

(This is authoritative - if examples are different they are in error).

```
Annotation => Object[{}],

AnnotatedFeature => Object[{
  attributes => {
    name        => UnqualifiedTypeName,
    type        => Type[Any],
    override    => { type  => Boolean, value => false, },
    final       => { type  => Boolean, value => false, },
    annotations => {
      type  => Undef[Hash[Type[Annotation], Struct]],
      value => undef,
  }
}],

AttributeKind => Undef[
  Enum[ constant, derived, given_or_derived ]
],

Attribute => Object[{
  parent     => AnnotatedFeature,
  attributes => {
    kind     => AttributeKind,
    value  => {
      type    => Any,     # constrained by attr type
      kind    => given_or_derived,
    }
  }
}].

Function => Object[{
  attributes => {
    parent   => AnnotatedFeature,
    type     => {type => Type[Callable], override => true }
```

```
  ]
}],

NameArray        => Undef[Array[Name, 1]],
NameOrNameArray  => Variant[Name, NameArray],
AttrMatch        => Hash[Name, Any],
AttrsOrAttrMatch => Variant[AttrMatch, NameOrNameArray],

# Invariants are described with Assertion objects
Assertion => Object[{
  attributes => {
    # A string that may contain interpolation of variable
    # values - strictly only ${varname} interpolations
    # where varnames are references to attributes
    #
    message    => String[1],

    # Defines (with defined_is) what it means to be
    # defined/undefined.
    # mutually exclusive with defined_is
    undefined_is => { type => Undef[Type], value => undef },

    # Defines (with undefined_is) what it means to be
    # defined/undefined.
    # mutually exclusive with undefined_is
    defined_is   => { type => Undef[Type], value => undef },

    # If assertion is a warning (deprecation, warning), or
    # an error.
    severity     => {
      type => Enum[deprecation, warning, error],
      value      => error,
    },

    # An identifier for this assertion (typically a mnemonic)
    # that can be used for internationalization of message,
    # tie an implementation to an assertion etc.
    assertion_code => {
      type => Undef[String[1]},
      value => undef,
    },

    # A declarative condition that enables the assertion
    # Can be true/false, or describe a match against the object's
    # attributes.
```

```
when       => {
  type => Variant[Boolean, AttrsOrAttrMatch],
  # enabled by default
  value = true
},

# Asserts that given attributes are undefined as per the rules of
# "what being defined" is.
assert_undefined => {
  type = Undef[NameOrNameArray],
  value => undef,
},

# Asserts that one of the given attributes are defined as per the rules of
# "what being defined" is.
assert_one_defined  => {
  type => NameArray,
  value = undef
},

# Asserts that none or one of the given attributes are
# defined as per the rules of "what being defined" is.
assert_none_or_one_defined  => {
  type => NameArray,
  value = undef
},

# Asserts that one or more of the given attributes are
# defined as per the rules of "what being defined" is.
assert_one_or_more_defined  => {
  type => NameArray,
  value = undef
},

# Asserts that two or more attributes having hash values
# have unique keys.
#
assert_unique_keys  => {
  # only meaningful across hashes (min 2 entries)
  type => Undef[Array[Name, 2]],
  value = undef
},

# Asserts that surface level values in given attributes
# are unique across all given attributes.
assert_unique_definitions  => {
```

```
    type => NameArray,
    value = undef
  },

  # Asserts that values in given attributes
  # are unique across all given attributes.
  assert_flattened_unique_definitions  => {
    type => NameArray,
    value = undef
  },

  # Asserts that the object has a certain state by matching
  # attribute values against matching values/types.
  #
  assert_attributes_match  => {
    type => Undef[AttrsMatch],
    value = undef
  },

  # Asserts that given values are names of attributes
  assert_attributes_exist => {
    type  => Undef[Variant[NameArray,
      Struct[{
        attributes => Variant[Name, Array[Name, 1]],
        type       => Undef[Type],
      }]],
    value => undef
  },

  # Name of function that is called to perform assertion beyond
  # the declarative assertions.
  asserted_by => String[1],
}
assertions => [
  { message => "defined_is and undefined_is cannot be both be set",
    assert_none_or_one_defined => [undefined_is, defined_is]
  },

  { message => "Attribute-reference to non existing attribute",
    assert_attributes_exist => [
      assert_undefined,
      assert_one_defined,
      assert_none_or_one_defined,
      assert_one_or_more_defined,
      assert_unique_definitions,
```

```
        assert_attributes_match,
        assert_attributes_exist ]
    },
}],



# This should be read type Pcore::Object = Object[...]
Object => Object[{
  attributes => {
    attributes => {
      type => Hash[Name, Variant[Type, Init[Attribute]],
      value => default,
    },
    functions  => {
      type => Hash[Name,
        Variant[Type[Callable], Init[Function]]
      ],
      value => default,
    },
    assertions => {
      type => Array[Variant[Assertion, Init[Assertion]]],
      value => [],
    },
    annotations => {
      # The Struct here is really Init[T] where
      # T is the type of the Annotation given as key, but Pcore
      # does not yet have this descriptive power.
      #
      type  => Hash[Type[Annotation], Struct],
      value => {},
    },
    equality => {
      type  => Undef[Variant[Name, Array[Name]]],
      value => undef,
    },
    equality_includes_type => {
      type => Boolean,
      value => true,
    },
    order_by => {
      type  => Array[Tuple[Name, Direction]],
      value => default,
    },
  },
```

```
  assertions => [
    { message =>
"equality cannot be specified when equality_includes_type is false",
      when => { equality_includes => false },
      assert => { equality => Undef }
    }),
    ]
}]
```

Object Type Invariants

- Attribute and function features share a namespace and their names must be unique in their parent object type, and the features inherited from the object types ancestors.
- Attributes and functions may not override each other.
- An attribute may override an inherited attribute
- A function may override another function to narrow its return type.
- An overridden function must accept the exact same parameters (neither more nor less)
- An overridden function may have different default values

TypeSet Specification

A `TypeSet` is at the root of a Pcore meta-model. (This is called a "Package" in Ecore, but that term was deemed to be to already overloaded in Puppet).

Here is the Pcore definition:

```
type Pcore = TypeSet[{
  name_authority      => "http://puppet.com/urns",
  pcore_version_range => "~2016",

  name                => "Pcore",
  version             => "2016.0.0",

  types => {

    # A relative qualified reference (no leading '::' allowed)
    QRef              => Pattern[/\A[A-Z]\w*(:?::[A-Z]\w*)*\Z/],
```

```
TypeName          => QRef,
SimpleTypeName    => Pattern[/\A[A-Z]\w*\Z/],

Annotatable      => Object[{
  attributes => {
    annotations => {
      type  => Undef[Hash[Type[Annotation], Struct]],
      value => undef,
    }
  }
}],

TypeSetReference => Object[{
  parent => Annotatable,
  attributes => {
    # The namespace authority. Defaults to the same
    # authority as the type set it appears in.
    #
    name_authority      => {
      type  => URI,
       kind => given_or_derived,
    },
    # The full model name of the referenced type set
    name    => QRef,

    # Range of compatible versions of the referenced type set
    version_range       => SemVerRange
}],

TypeSet => Object[{
  parent => Annotatable,
  attributes => {
    # A URI that anchors the identities of named elements.
    # This can be read as the answer to "Says who?"
    #
    name_authority       => URI,

    # Full type set model name - the prefix of all contained
    # modeled types
    name                 => TypeName,

    # The version of this type set
    version              => SemVer,

    # The authority defining the Pcore used to define this
```

```
  # type set, if different than the default official Pcore.
  #
  pcore_uri            => {
    type  => URI,
    value => "http://puppet.com/urns/pcore"
  },

  # The version range of Pcore required to understand
  # this type set.
  pcore_version        => SemVer,

  # Definitions of types in this type set.
  types                => Hash[TypeName, Type],

  # References to other type sets used by this type set
  # made available under an alias simple name that does not
  # clash with any other type in this type set (often the
  # last part of the referenced type set's namespace).
  #
  # For example:
  #
  #   name_authority      => "https://www.github.com/userx",
  #   name => "Garage",
  #   references => {
  #     Cars =>   {
  #       name          => MyOrg::Cars,
  #       version_range => ">=1.0.0 <2.3.0"
  #     }
  #   },
  #   attributes => {
  #     example_car => Cars::Cabriolet
  #   }
  #
  references => {
    type  => Hash[SimpleTypeName, Init[TypeSetReference]],
    value => {},
  }
},
equality => [
  name_authority,
  namespace_prefix,
  version
],
assertions => [
  { message =>
```

```
            "An alias for a referenced type set must be unique",
          assert_unique_keys => ['./references/alias',
'./attributes/name'],
        }
      ]
    }]}]
```

The TypeSet's Attributes

name

The `name` is the fully qualified technical name of the TypeSet in the Pcore/Puppet namespace. All the types in the type are inside this namespace.

When a type set is used, it may be bound to an implementation that is "mounted" on this namespace, but this is not an absolute requirement since:

- a target platform may use a different naming scheme, in which case the actual name may be a transformation of the namespace prefix + type name.
- an implementation may need more than one version of a type model at a given time (for example, to transform between the two versions)
- the name may clash with something already existing and the implementation is mounted on a completely different namespace.

As an example the type set named "Pcore" would typically be mapped to the Ruby name "Puppet::Pcore", while in Java it may be mapped to "com.puppet.pcore".

Note the colloquial name is not part of the technical name. It is typically used when mounting the type set in a namespace that is different than the namespace_prefix, but this is by convention only. There is nothing stopping someone from mounting Pcore under for example Veggie::Peacore or X::Y. Naturally talking about the model and reading code then becomes much harder, so mappings like these should be avoided unless there is a very good reason).

name_authority

The `name_authority` is a "well known URI" that together with the `name` (and version) fully defines the identity of this type set. The `name_authority` is a URI that identifies a namespace that is guaranteed to be unique by a very high degree of certainty. Typically a http/https URL should be used as the domain has ICANN as name authority. A path in this URI can indicate some sub authority (like a division/department within an organization). This "sub authority" can

simply be a path. Individuals who does not have their own domain can use a domain where they have a registered identity, for example https://github.com/hlindberg, or if the individual defines models in disparate domains in different repositories at github to a repository under this user's control.

The name_authority's only purpose is to ensure that the used name is globally unique with a reasonable degree of certainty without requiring registration with a Pcore specific name authority. Pcore does not perform any comprehension of what is in this URI other than using it as an identifier in string form. Tooling that performs model to model transformations may use it as some modeling technologies may have defined semantics for this URI (it may for example contain the model name and version in some technologies).

Specific to puppet modules, a user may want to use the URL that identifies a publisher on the Puppet Forge - for example https://forge.puppet.com/aptituz. The `name` should then start with the name of a module. (This is compatible with how the current Puppet autoloader works, so should not be strange to users).

The `name_authority` URI should by convention be available as a document/page when using the URI as a URL, but this is not an absolute requirement.

URIs that are not controlled by a name authority should be avoided. The URIs myscheme:foo and file:///users/bob are very bad because they mean nothing to other users, and users on a different host than where bob is a user.

Such URIs are however of value in an implementation when name_authority URIs are mapped to something that exists in that specific environment. Any such mapping is however specific to an implementation of Pcore on a given target. (As an example, in a particular running instance that makes use of a Garage model defined by https://github.com/hlindberg, may be locally avaiable in file:///repositories/github/hlindberg/garage/model/garage_1_1_0.pcore).

pcore_version

The `pcore_version` specifies which versions of Pcore that this type set is compatible with. Or in other words, which versions of Pcore that a user of the model must have an implementation of to be able to read and fully understand the type set.

Since Pcore is defined in itself, it looks a bit strange that it requires a version of itself. This has a practical use though - even if newer versions of Pcore introduces new features, those features may be expressed in an earlier version of Pcore in order for older implementations to be able to load and use that definition.

pcore_uri

The `pcore_uri` identifies that this is a Pcore model. The `pcore_version` expresses a version of the definition of what Pcore is. By default, *Pcore is Pcore as defined by itself*, and this definition was created by Puppet Inc on the well known URN http://puppet.com/urns/pcore. Thus, there is no reason to specify this in every use of Pcore.

However, since we follow the principle of "no magic should be reserved", it must be possible to fork Pcore itself and give it a new definition. While this fork could be given a completely different name, it would mean that no reference to it as "Pcore" would work. Also, the stewardship of what Pcore is could change over time - it is not unheard of that technologies that were created by a corporation ends up being under control of some other corporation (acquiring the original creator), or that the stewardship is spun off to an open source organization or consortium). Last but not least, the original steward could change name (luckily, the specification of Pcore was not published before Puppetlabs changed name to Puppet).

In an implementation the Pcore model is typically bound to the specific implementation of Pcore available on that target. Such an implementation may handle only one version of Pcore, have separate implementations for multiple versions, or an implementation that understands multiple versions by creating dynamic implementations.

references

The `references` maps types in other type sets into names that can be used in this type set. A reference to another model is made with an instance of `TypeSetReference`. The references attribute is a hash that maps local type names to referenced type sets. The TypeSetReference contains a `name_authority`, a `name`, and a `version_range` that together specifies a type set. The alias is always used when referencing a type in a referenced type set.

The aliasing is a powerful concept that reduces the amount of refactoring required if a model shifts from using a variant or version of a model where there has been a change in the namespace prefix. It also disambiguate if there is already a type in this type set's namespace that would clash, for example if this type set is `Example::Cars`, and a type in this type set is `CarTypes::Cabriolet`. There is then a desire to use the external model `CarTypes`, and it also defines `Cabriolet` (together with a bunch of other car types; SUV, Sedan, etc.). Without the aliasing, a name like `CarTypes::Cabriolet` is ambiguous (although told apart by the `name_authority`, it is not included in names in the Pcore domain). We therefore use an alias that is unique among the names already in use (or anticipated) in this type set, for example we may use the name of the publisher, or something like `ExtCarTypes`. We can then differentiate between `CarType::Cabriolet` (defined in this type set), from the "imported" variant which we can reference with for example `ExtCarType::Cabriolet`.

Aliasing is important, because both type sets may be mounted on an implementation under some other name, and we cannot in a model know any absolute fully resolved names a-priori.

Referenced type sets are not externally available via the referencing type set; if `Foo` references `Bar` under the same alias, it is not possible to access `Bar` via `Foo::Bar` outside of `Foo`.

equality

The equality of a TypeSet consists of the attributes `name_authority`, `name`, and `version`.

Pcore Data Types

The data types in Pcore are those in the Puppet type system. Extending the set of basic data types requires a new version of Pcore.

The Puppet Type System types are shown at the beginning of this document.

Type Set Factory

When a Pcore modeled object is to be created in an implementation of Pcore there is the need to map the Pcore implementation agnostic format to an implementation. As an example, in the Ruby version of Puppet, the AST object `ArithmeticExpression` is implemented with RGen (Ecore based modeling in Ruby), and the Locator object used in the AST is a plain old ruby class. When these are serialized and instantiated there must be a definition of how this mapping takes place.

Naturally the factory in a C++ implementation is different than that of a Ruby implementation. Each implementation must therefore have a mapping from modeled type to runtime type. This model can also be expressed with Pcore.

Pcore allows a type alias that associates a Runtime type with a Pcore type. This association is bidirectional and is used to perform a mapping from implementation specific classes to Pcore types and vice versa. The mapping is typically pattern based, but could also be a static definition of a single implementation type to a Pcore type.

As an example, this maps all types in the Puppet AST implemented in the ruby module `Puppet::Pops::Model`:

```
type Runtime['ruby',
  [
```

```
    /^Puppet::Pops::Model::(\w+)$/,
    'Pcore::AST::\1']
  ] = [
    /^Pcore::AST::(\w+)$/,
    'Puppet::Pops::Model::\1'
  ]
]
```

(While quite opaque this is something that is used by the implementor of a model when the implementation is in a language other than puppet).

The left hand side is the implementation/runtime side and it defines the mapping to Pcore. The right hand side defines the mapping in the reverse.

For any given Pcore `namespace_uri`, there are additional factory model URIs that is derived from the namespace URI. Thus for the namespace `http://www.puppet.com/examples/ExampleAST` there exists one factory namespace URI `http://www.puppet.com/examples/ExampleAST/Factory`. This namespace URI is naturally mapped to different concrete mappings depending on runtime implementation. How this is done is implementation specific - the runtime may know how to handle only certain models (a priori), or it may use Pcore to define the mapping dynamically, or it may have used a Pcore mapping to generate code etc. For an implementation that uses a mapping to a dynamc Pcore defined runtime mapping (say for Ruby, it may map to the URI `http://www.puppet.com/examples/ExampleAST/Factory/Ruby/v4.4` which, when resolved to a file may refer to an `ast_factory_map.pp` somewhere on the file system. That file would then contain bidirectional mappings as exemplified earlier.

Dynamic Pcore Implementation

It is not required to have specific implementations of everything in a Pcore model. This is only required if there is behavior associated with an object and the implementation needs to invoke such operations. Thus it is perfectly fine for an implementation to have a generic and dynamic implementation of a Factory that simply creates instances of some suitable wrapper class and say a hash to hold values of attributes. This wrapper would have generic getters (and possibly setters in a language where mutation is allowed). These getters and setters are then guided by the Pcore type information that defines the properties of the values (validity etc.).

As an example, a database such as PuppetDB would not be interested in the actual behavior, it would simply record the data for the purpose of later serving it back, and it can then use a

generic/dynamic implementation. A client of the database may retrieve the data, have a limited implementation of some of its operations (required to perform calculations say).
Other examples are for visualization where it is enough to get the attributes (for example JavaScript) in order to draw nice graphs.

Model Set

A Model Set defines the context in which serialization/deserialization and resolutions of references take place. It is implementation specific and may be hard coded to deal with only a priori defined models, or it may be defined dynamically in terms of other models.

Every model set has a reference to a Pcore meta-model. As you may recall, a meta-model is represented by a Pcore type set, and the type set is associated with a factory. This is the mechanism with which each model can make reference to things in Pcore, for example `Tuple[Integer, String]`, when such a Type Reference is obtained it must be turned into an object on which it is possible to operate. Via the factory, that may have been set up with a factory model mapping - the type reference will be transformed into an instance of `Puppet::Pops::Types::PTupleType`, with its types array attribute resolved to contain a `Puppet::Pops::Types::PIntegerType`, and `Puppet::Pops::Types::PStringType`. If we go the other way, and serialize something via the Model Set, the reverse operation takes place.

There are some defined model sets:

- the global "puppet compilation" - what is in effect when puppet code is being evaluated in general during a catalog compilation
- "RPC" call between collaborating processors consists of a call request model set, and a call response model set; that together define the bidirectional data transformations required.
- A programmatic model set constructed by an application (or built in puppet feature) - for example the parser using XPP to avoid parsing.

The context, implementation and usage will be defined separately as they are part of an implementation for a particular target.

Document

A document is a container of a serialized Pcore model. Source code form is considered to be one serialized form and it is the default when reading a .pcore document. A 'Document' is however a general concept.

It is expected that an implementation may have different kinds of Document types implemented. In general a Document type (such as the Puppet XPP format) is defined by a Pcore model that defines its content and mime type. There is no requirement that Document type handling application in any language is implemented by using Pcore - its implementation could simply have been directed by what the model for it dictates.

See Serialization Concepts for the default Document in serialized form. There may be other parsers that directly read some kind of textual document / serialization format and directly maps that into an instance of an object (-graph/tree) that complies with Pcore without first transforming it to Pcore. Such documents in "serialized form" would have different envelops (or no envelope at all if document type is derived only from file extension; which would be quite typical).

```
    DocumentType => Object[{
    attributes => {
      # Defines what the content is - if multi rooted,
      # the type should be Array[Variant[...]] or similar.
      #
      content_type => {
        type => Type,
        value => NotUndef,
      },
      # The base mime type without encoding (the +msgpack
      # or +json encoding part should not be included)
      mime_type => {
        type => String,
      },
      version => SemVer,
      document_type => String,
      parameters => {
        type => Array[Name],
        value = [version, document_type]
      }
    }
  }]
  }
}]
```

Serialization Concepts

Serialization (used in this document to denote both serialization and deserialization unless clear from context that it only refers to the "object to on-the-wire" process).

Tabulation

Tabulation means that a serializer replaces common data values with a reference. The data value only appears once in the serialization. This is very valuable as it speeds up both serialization and deserialization as well as reducing the amount of memory needed when the deserialization has built the finished result.

Tabulation means that the result is not readable by humans even if a text format is used; the result is simply too complex. Therefore Tabulation is only used for machine to machine M2M communication. (You can naturally convert a M2M format to a format in human readable form).

Tabulation is efficient when values repeat. To make tabulation work for a larger set of data types it is of value to break up Path, URI, and Qualified Reference/Name values into segments. This also enables further compaction as there may be a large amount of values that share a sequence of common segments.

Tabulation is typically implemented in serialization by writing and assigning an index on first occurrence, and using the index on each subsequence occurrence of the same value.
The deserializer does the tabulation in reverse. For tabulated slots it remembers the value and gives it an index. When expecting a value and getting an index it looks up the value.

Envelope

An envelope wraps a serialization into something that describes what the serialization/content is. Two parties exchanging information may do so by separating the envelope from the data stream, or the may be communicating in such a way that they have no direct contact and the envelope and content must travel together (envelope containing data, or side-by-side).

One example is a REST request where the envelope is in the request (I can handle these kinds of data)/response (this is the data format you are getting it in). Another is saving the content to a file - somehow the envelope needs to accompany the data; as a manifest in a zip file, as a mime multipart stream, as a REST multipart. etc.

The Pcore envelope is simply a MIME type.

```
PcoreSerialization : Envelope Payload ;

Envelope
  : MimeType BlankLine
  ;

MimeType
  : PcorePart '+' BaseEncoding (';' Parameter '=' Value)*
  ;

PcorePart
  : 'application/vnd.puppet.pcore'
  ;

# The main machine to machine format is msgpack. The json format
# is intended for human use and alternative representations where
# msgpack is unsuitable.
# Other formats may be added over time.
#
BaseEncoding
  : 'msgpack'
  | 'json'
  | # extended as needed with additional encodings
  ;

Parameter
  : 'document-type'
  | 'name_authority'
  | 'version'
  | /[\w-]+/   # other parameters as directed by a particular document-type
  ;

Value
  : /[\w-]+/
  ;

# A blank line is used as the delimiter as specified by the MIME RFC
# it enables delivering mime type parameters in the future without
# requiring that they are all on one line.
#
BlankLine
  : '\r'? '\n' '\r'? '\n'
  ;

Payload: <OPAQUE-MIME-TYPE-DATA> <EOF>
```

```
  ;
```

As an example - the XPP format (serialized Puppet Language parse result) is written to a file, As shown in the following example:

```
# application/vnd.puppet.pcore+msgpack
; document-type=XPP
; name_authority=http://puppet.com/urns
; version=1.0.0

<OPAQUE-MIME-TYPE-DATA>
<EOF>
```

In practice an implementation may cheat and only include the very first like if by context/convention it is known that files with the extension .xpp can only mean the kind of xpp that is defined by "puppet.com". Since that is not universally true (anyone can write anything into a file and give it an .xpp extension) to be completely clear, all three attributes should be given.

document-type

The document-type defines the expectancy what the deserialized content is. The document-type together with the namespace-authority and the version completely identifies what the expectations are since they form a reference to a Pcore model that defines a Document type.

```
TypeSet[{
  name_authority  => 'http://puppet.com/urns',
  name            => 'XPP',
  version         => '1.0.0',
  # etc
  references => {
    AST => {
    name => Puppet::AST.
    version_range => '>= 4.5.0',
    }
```

```
  }
  types => {
    xpp => Object[{
    parent => Pcore::Document,
    attributes => {
      content_type => {
        kind     => constant,
        override => true,
        value => AST::ParseResult,
      },
      mime_type => {
        kind => constant,
        type => String,
        value => "application/vnd.puppet.pcore"
      },
    }
  }]
  }
}]
```

Streaming Ability

It is beneficial if a serialization format is streamable; that is, that processing of large entities can start as soon as there is some data rather than having to wait for all data before seeing any of it.

As an example a JSon serializer will read and return complete arrays, and complete hashes; there is no way to get a stream of the elements that build up these structures. In practice this means that the entire serialization must be read into memory. And if this represents an intermediate form; yet another copy of the transformed data must be brought into existence in memory before the loaded deserialized data can be released.

Further, if the length of a serialized element is not known up-front, the system will have to allocate and resize containers dynamically with wasted memory and performance as a result because of copying/moving and creating garbage.

Data Representation in Streams

All serialization technologies have a core set of base data types. Some have extension mechanism which makes it easy to represent additional data types (for example MsgPack). In

other formats (for example Json/Yaml), one of the existing data types must be used to describe the extensions (and then it can no longer represent that data type directly - it must then also be encoded as an extension).

A concern when designing this is efficiency - especially for the textual formats.

Considering the streaming ability (that is up-front information about sizes, and tabulation) we need a handful of instructions:

- define value - i.e. assign value to index
- use value - i.e. given an index of an already defined value, produce that value
- start container of type having a given length
    - array of length n
    - hash of n elements
    - object of n attributes
- extended data type T - plus a representation of it using the available data types in the particular serialization format
- literal value in the particular serialization format (if it conforms with the definition of the serialized data type)

Exactly how this is done is up to the specification of each format. Here is an illustration of what it can look like.

extension number	**meaning**	**payload**
0	define value	id and one serialized entry
1	use value	id of defined value
2	start array	count, followed by count number of serialized entries
3	start hash	count, followed by count number of key/value tuples
4	start object	count, type, followed by count number of key/value tuples
...		
10	literal default	no payload
11	literal regexp	payload is a string
...	etc.	

puppet data	serialized as Json (illustration of principle)
`1`	`1`
`[a, b, c]`	`[2, 3], 'a', 'b', 'c'`
`{a => 1, b => 2}`	`[3, 2], 'a', 1, 'b', 2`
`"hello"`	`"hello"`

`[Path('/tmp/foo'), Path('/tmp/bar')]`	`# An array with two entries`
	`[2, 2]`
	`# entry 1`
	`# object, 'Path' (defined as 1)`
	`# having 1 attribute`
	`[4, [0, 1, 'Path'] 1]`
	`# The attribute 'segments' (defined as 2)`
	`[0, 2, 'segments'],`
	`# 'segments' is an array with two entries`
	`[2, 2]`
	`# The first entry is 'tmp' (defined as 3)`
	`[0, 3, 'tmp']`
	`# The second entry is 'foo' (defined as 4)`
	`[0, 4, 'foo]`
	`# entry 2`
	`# Also a path (using definition 1)`
	`[4, [1, 1], 1]`
	`# attribute 'segments' (using definition 2)`
	`[1, 2]`
	`# The path part 'tmp' (using definition 3)`
	`[1, 3]`
	`# The path part 'bar' (defined as 5)`
	`[0, 5, 'bar']`
	`# END OF STREAM`

Different serialization technologies have different mechanisms. As an example MsgPack has extensions that efficiently combine extension with value - compared to the above illustration most of the special markers are single bytes as opposed to arrays.

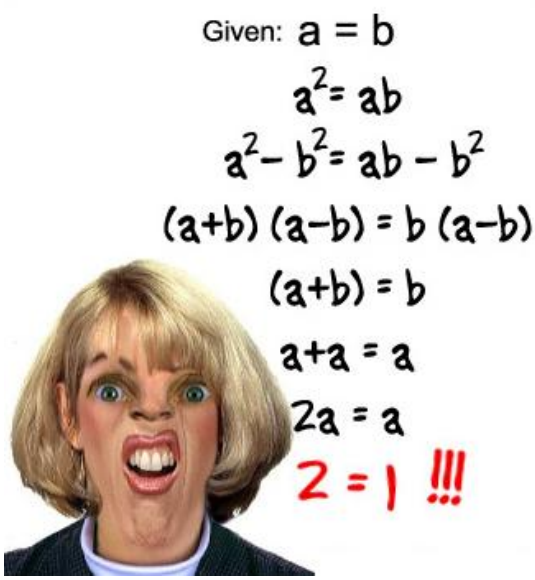
The Modelers Creed

This is my model. There are many like it, but this one is mine. It is my life. I must master it as I must master my life. Without me my model is useless. Without my model, I am useless. I must use my model true. I must abstract clearer than the enemy who is trying to confuse me. I must make clear before my enemy confuses me. I will. My model and I know that what counts in computing is not the lines we type, the bugs we kill, or the self promoting tweets we make. We know that it is the clarity that counts. We will be clear.

My model is human, even as I am human, because it is my life. Thus, I will learn it as a sister. I will learn its weaknesses, its strengths, its parts, its accessories, its features and its power. I will keep my model clean and ready, even as I am clean and ready. We will become part of each other.

Before Turing I swear this creed. My model and I are the defenders of my realm. We are the masters of our enemy. We are the saviors of my life.

So be it, until victory is ours and there is no WAT.



TODOs

- check that 'final' support is specified everywhere
- value_reference - using an XPath to pick/compute a value from elsewhere (for constant/default value)
- The notion of 'abstract type' is missing - now anything can be instantiated
- Different kinds of attributes or types / i.e. extensions to the model (parameter/property distinction in resource types. Extend model, or model as annotations. Is a classification concept enough? I.e. a super type can introduce classification available in features in subtypes.
- Object references vs. Containment
- ModelSet / resolution contexts (general, a use-case/app, a call/return site etc).
    - when does an interface become an implementation you can use
    - a type set under <root>/types - mapping to impl - or something else?
    - protection against wild and crazy impl mappings (worst case addressed already, but can still do weird things with that).
- Equality rules allow derived values to be included. This is problematic when using a dynamic implementation (where the derived logic is not present) as it alters the rules for equality. If this ability is denied we must also deny override of a non derived attribute with a derived if this attribute is part of equality.
- For resource type it is of value to be able to express cross resource assertions - see https://tickets.puppetlabs.com/browse/PUP-6277 - much in the same fashion as it is of interest to do mutual exclusion rules in an object instance. This could perhaps be modeled as Assertion objects with xpaths that operate on the catalog. A resource type would need to deliver these assertions- maybe as an annotation? CatalogAssertions(...). The compiler knows to look for that annotation and then adds those assertions to a full set that is run on the catalog at the end of the compilation.
- Internationalization - the use of an error code is not good enough, and different from other parts of puppet - consider using full message text as key. Also needs to define how validation can use an I18N package with message text.

Short form new from PUP-7853 should be added to this document:
Before this commit, a type alias of a Object type would have to be

declared like this:

type MyType = Object[{ ... }]

with this commit, the brackets are optional for the Object type so that

the above expression can be written:

type MyType = Object { ... }

Actually, the word 'Object' can also be left out so this is also legal:

type MyType = { ... }

The 'Object' can also be replaced by something else, in which case it

becomes the parent type. Thus, this description:

type MyType = Object[{ parent => ParentType, ... }]

can now be written like this:

type MyType = ParentType { ... }

History of changes:

- `namespace_authority` changed to `name_authority`
- `TypeSet` references changed syntax from using an array where alias was an attribute of the reference to instead be a hash where the key is the alias.