Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Predicate Collection Classes

Cristina Videira Lopes[a] Rohan Achar[a] Arthur Valadares[a]

a. University of California, Irvine, USA
http://www.ics.uci.edu

Abstract Collections are at the heart of every program. Modern programming languages have acknowledged this fact by including increasingly better expression mechanisms for manipulating collections of data and objects. When existing objects are selected as elements of collections, often there is an implicit intention that those objects, by means of having been selected, take on new roles. Such is the case, for example, with shared data in distributed simulations, which changes over time, and which distributed components may use for different purposes as the simulation unfolds.

This paper presents the concept of Predicate Collection Classes (PCCs). PCCs are both classes and specifications of how to select and reclassify objects from existing collections. We describe the informal semantics of PCCs and demonstrate how they can be used to express several iterative algorithms that make heavy use of collections. Finally, we summarize our use of PCCs in a framework for distributed simulations.

Keywords collections, class, reclassification, relational algebra

1 Introduction

In many branches of study, including mathematics, the word *class* is, formally or informally, used to denote sets of things that can be unambiguously defined by a property that all of its members share [Lev79]. In this sense, *classification* is the process of placing data with common properties into sets or collections. For historical reasons, in OOP languages, the word *class* has a related, but slightly different, meaning: a *class* is a template for *constructing* objects that have one property in common, namely, they are modeled after the same nominal pattern of state and behavior, the *class* itself [Bru02]. In OOP, we can look at a class C as a function that takes data and returns an object with a specific set of fields and methods, as defined by some pattern; all objects instantiated using C are said to be of the same class.

We are interested in expression mechanisms that allow us to classify data depending on runtime conditions, or *predicates*, associating it with behavior dynamically. We use the words *classify* and *classification* to denote both the identification of data with certain properties (i.e. the broader meaning) and the process of associating that data with certain pre-existing patterns (i.e. the OOP meaning). The idea is that of *predicate collection classes* (PCCs, from here on), those whose extents are

 In Journal of Object Technology, vol. 16, no. 2, 2017, pages 3:1–38. doi:10.5381/jot.2017.16.2.a3

automatically determined by a predicate, rather than being explicitly manipulated, and whose behavior is given by [OOP] classes. Such expression mechanisms are extremely convenient, especially in long-lived, complex applications where the data changes dynamically and may need to be reclassified often, such as simulations [Fuj00] and the processing of streaming data [Mut05].

As a motivating example, consider two functionally independent, but data dependent, simulations: a traffic simulation and a pedestrian simulation. The traffic simulation moves cars along roads, performing collision avoidance between them, using a rich model of cars and roads. The pedestrian simulation moves pedestrians in sidewalks who can, at points, cross roads. The process of detecting possible collisions between cars and pedestrians establishes a data dependency between the two otherwise independent simulations: information about cars must flow, directly or indirectly, to the pedestrian simulation, or vice-versa.

There are several alternatives to model this situation. One option is for the car simulation to import/access, at every simulation step, the entire collection of pedestrians from the pedestrian simulation; another option is for data to flow the other way around. None is these options is ideal, for two reasons: (1) the vast majority of cars and pedestrians are not at risk of colliding, so it will be wasteful to replicate/access them all when only a *subset* of them is of interest; and (2) the information that the car simulation needs of pedestrian objects, or that the pedestrian simulation needs of car objects, is only a small portion (i.e., a *projection*) of the potentially rich data models used by the respective simulations – essentially only their positions are needed. A better alternative is for one of the simulations to only import/access portions of the other simulation's objects that are within a certain range of its own objects. But, in this case, an even better alternative is to not import or access the other simulation's objects at all, and, instead, rely on an external substrate of all data that can help model and classify the internal data more expressively. For example, the pedestrian simulation can model the concept of *pedestrian in danger*, which corresponds to the subset of its own pedestrians that are at a certain distance of any car in the other simulation. In doing so, no car flows explicitly into the pedestrian simulation; the collection of *pedestrians in danger* is simply a subset of the existing pedestrians predicated on state that exists on another collection elsewhere. Pedestrians in danger behave differently than all other pedestrians, for example, they may move faster or stop. Once they move out of danger ranges, they become regular pedestrians again.

```
1  pclass PedestrianInDanger(Pedestrian pedestrian, List<Car> cars) :
2      predicate : :
3          foreach c ∈ cars :
4              if pedestrian.near(c) :
5                  return True
6              end
7          end
8          return False
9      end
10     method avoid() :
11         // Move the pedestrian out of the road!
12     end
13 end
```

Algorithm 1: Potential Pedestrian and Car collision avoidance code.

The pseudo-code in Algorithm 1 sketches the main idea of what we want to achieve: we want to create and classify collections of objects automatically from other collections. In this case we want the collection class `PedestrianInDanger` to contain `Pedestrian` objects predicated on a given list of `Car` objects, and we want this predicate collection class to have its own specific methods – in this case, method `avoid`.

Proposals for how to achieve similar goals can be traced to the early days of OOP, and include predicate classes [Cha93], multiple most specific classes [BG95], dynamic reclassification of objects [DDDCG01], and dependent classes [GMO07]. We take a fresh look at this idea, placing it in the context of modern applications and frameworks. Two characteristics make our approach different from what has been proposed before: (1) our focus on *collections* of objects (predicate classes), rather than on construction of individual instances (attributive classes); and (2) our approach to handling object state changes.

In languages with mutable state, PCCs pose many challenges, some of which are critical to both the semantics and the implementation of the concept. An object that is classified as an instance of some predicate class C at one point in time, may see its internal predicate become invalid some time later. In the example above, a call to the method `avoid` (line 10) may change the position of the pedestrian in danger, violating the property that made that object exist in the first place. From that point on, it is unclear what should happen to that object. Should it cease to exist? Should it continue to exist but in a zombie state? Should it continue to exist within a certain scope, but with the understanding that the predicate may be invalid? We chose this latter option, as it is the closest to the semantics of modern collection classes. This is will be explained in more detail later on.

We present a programming model based on collections of objects created from other collections of objects, where the classification predicates are declared by static queries defined in classes, rather than being intermingled with the rest of the program logic. Although this programming model can be used in many situations, it is designed specifically for applications where the input data changes as the application runs, requiring frequent (re-)classification.

Our work builds on many ideas that came before related to dynamic reclassification of objects. The main contribution is the design of **a single mechanism that serves two goals simultaneously**: selecting objects from collections and dynamically associating their data with specific behavior, effectively making them be objects of additional classes.

It should be noted that this paper describes design work that is implemented, publicly available, and used by the authors in non-trivial applications, but whose main tenet – that the design is beneficial for software development – is not yet assessed systematically. This paper focuses on the design itself, and the motivations for it; a systematic evaluation of its benefits will be the topic of another study.

This paper is organized as follows. Section 2 gives an overview of predicate collection classes and how they can be used to create object frames. Section 3 describes object reclassification in more detail and Section 4 presents the complete list of algebraic operations that underlie PCCs. Section 5 shows how our programming model can be used by data processing applications. Related work is presented in Section 6. Finally, Section 7 discusses open issues with the design and concludes the paper.

Decorator	Target	Origin
@subset($Class$)	class	PCC
@projection($Class, Field_1, ..., Field_m$)	class	PCC
@join($Class_1, ..., Class_N$)	class	PCC
@union($Class_1, ..., Class_N$)	class	PCC
@intersection($Class_1, ..., Class_N$)	class	PCC
@parameter($Class, mode$)	class	PCC
@dimension($Type$)	property	PCC
@$Property$.setter	property	Python
@staticmethod	method	Python

Table 1 – Summary of decorators used by PCCs.

2 Predicate Collection Classes: Overview

This section gives an overview of the main design elements of PCCs using as an example the simplest of all operations, subsetting. Given that our first implementation of PCCs is in Python, the examples are written in Python. The PCC capabilities are embedded without changing the language, using our own decorators as well as some pre-existing ones. Table 1 summarizes these decorators, the elements they apply to, and whether they are our own or Python's.

In general, PCCs are collections of objects created from one or more collections of objects of certain classes or types, for which certain predicates hold. A PCC defines both a class and a collection of instances of that class. Listing 1 shows one example where the PCC ActiveCar is defined as a subset of instances of the regular class Car.

This first example lends itself to a few observations, all of which are applicable to all PCCs, not just subsets.

- The data of the elements in these collections is given by properties tagged as **@dimension** (see lines 2, 5, 8).[1] The set operations are established by declarations immediately above the class declarations, in this case **@subset(Car)** (line 13). Other operations will be introduced in the next section.

- Besides the operation declaration (in this case, Subset), the other user specification pertaining to PCC is the predicate, **___predicate___**, a static method, which, in this case, establishes that a Car c is active if its Velocity field is neither the zero vector nor None (line 16). The arguments for predicate depend on the operation declaration, and this is enforced by the PCC language processor; in the case of subsets, there is only one argument, one whose type is that of the subsetting declaration above (in this case, Car). This allows the programmer to define the filter for determining which elements of the original set are to be selected.

- PCCs construct instances of their type, not of the type of their supersets. In this case, the ActiveCar PCC will create instances of ActiveCar, not of Car. In other words, PCCs are not just about selecting objects from lists and returning the objects that honor in invariant; for each object that honors the predicate, a new instance of the PCC class will be created.

[1] For simplicity sake, we omit the setter methods in these examples, but they need to exist if the values of properties are to be changed.

```
1  class Car:
2    @dimension(int)
3    def ID(self): return self._ID
4
5    @dimension(list)
6    def Position(self): return self._Position
7
8    @dimension(list)
9    def Velocity(self): return self._Velocity
10
11   #...methods of Car...
12
13 @subset(Car)
14 class ActiveCar:
15   @staticmethod
16   def ___predicate___(car): return not (car.Velocity == [0,0] or car.Velocity == None)
17
18   def Move(self):
19     self.Position[0] += self.Velocity[0]
20     self.Position[1] += self.Velocity[1]
21     if self.Position == (100, 100): self.Velocity = None
22
23 cars = []
24 # ...fill in the list of cars...
25 while (True):
26   foreach aCar in pcc.create(ActiveCar, cars) as aCars:
27     aCar.Move()
```

Listing 1 – Subsetting

- PCCs can define their own fields, properties and methods. In the example above, the PCC ActiveCar defines a method Move() (lines 18–21) that doesn't necessarily exist in class Car (if it exists, it is an unrelated method). In this sense, PCCs are regular OOP classes, with their own behavior that is independent of the original objects' classes.
- PCCs acquire dimensions (i.e. properties) defined in the original classes of their supersets. In this case, ActiveCar acquires all the dimensions of class Car; that can be seen in the body of method Move, which refers to the fields Velocity and Position defined in class Car (lines 18–21). This mechanism is not inheritance, even though for subset operations, in particular, the dimension acquisition semantics is very similar to inheritance. For other operations, the differences will become much clearer in the following sections.
- Lines 23–26 show how PCCs are created and used: a function `pcc.create` takes a PCC class name (in this case ActiveCar) and a collection of objects from which to create reclassified instances, and returns those new instances.

The next two sections cover object reclassification and the different kinds of PCCs, respectively, in more detail.

3 Object Reclassification

In order to explain object reclassification, we will use the example of the previous section involving collections of cars and subsets of active cars (see Listing 1). We now focus on the bottom part of that snippet, the infinite loop (lines 25–27). Within that loop, in each iteration, a collection of reclassified objects is created given a list of cars (line 26). The collection aCars is a subset of the collection of cars that honor the predicate defined in ActiveCar. For every active car in this collection, the method Move is invoked (line 27); that method, which does not exist in class Car, changes the active car's position (lines 18–21), and may change its velocity to None (line 21). Those changes may or may nor be propagated to the original car instances, depending on whether copy or reference semantics is used; both semantics will be explained next. In either case, in the next iteration, the next reclassification will get a new collection of active cars.

3.1 Object Model and Reclassification

Figure 1 illustrates the required object model upon which PCCs can be defined. The creation of a class instance, for example `aCar = Car()`, results in two components being created: a set of fields that hold the object's state and a set of methods that access that state. The object state exists independently of the methods, and is a first-class object in itself – i.e. it can be referenced.

Several object-oriented languages provide this object model, or some version of it; Python is one of them. Other languages such as Java and C# do not expose the object's state as first-class, but the set of fields and properties of the object can be accessed via reflection, which make it possible for PCCs to be implemented in those languages too.

PCC objects go through a process of reclassification, which means that they are instances of different classes than their originals. In the Cars example (1), in line 28,

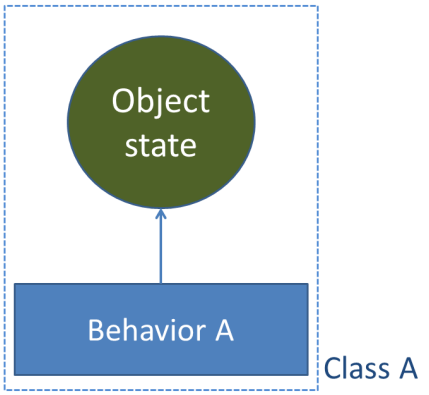

Figure 1 – Object model: an instance of a class is a combination of state and behavior (methods). Different behavior may be dynamically associated with the same instance state.

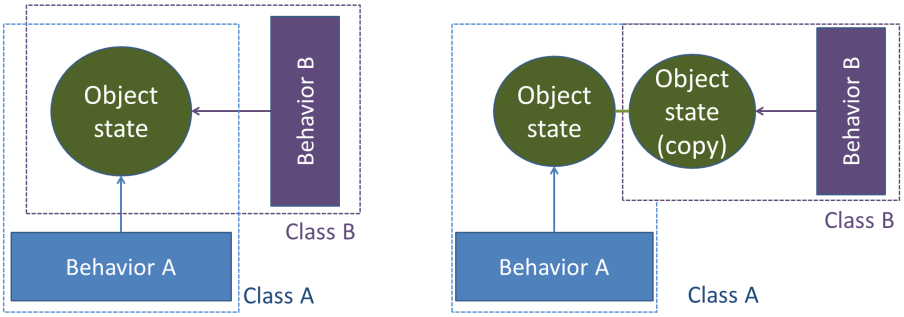

Figure 2 – PCC creation by reference (left) vs. by copy (right).

the car instances that honor the predicate (line 16) will be available as instances of ActiveCar. The relation between the reclassified instance and its original depends on whether a reference or copy semantics is used, which is the topic of the next subsection.

3.2 Reference vs. Copy Semantics

There are two mechanisms by which reclassification can be achieved: by referencing the state of the original object directly, or by copying data between the original and the reclassified instance. Figure 2 illustrates the two semantics. Both are possible, and we have experimented with both, but ultimately, we decided to adopt a reference semantics for the core of PCCs (Figure 2, left). Nevertheless, here, we discuss the two possibilities, their strengths and weaknesses, and the context where copy semantics is required.

If the state is used by reference, the PCC object and the original have different types (and therefore different interfaces) but point to the same object state. State changes can be done via any of the interfaces. In Python, we achieve reference semantics between the state of different instances of different classes simply by setting

the ___dict___ field of reclassified instances to that of their original counterparts.

If the reclassification is done with a copy semantics, then the creation of a PCC involves making deep copies of the objects' state at that point in time. From then on, the state of the PCC instance becomes independent of original. In order to prevent copying more than necessary, only attributes marked as dimensions are copied. Additionally, dimensions that are object references are followed and copied recursively. The copying process keeps track of which objects have been ingested, so that no two copies of the same object are included.

Reference semantics provides true multi-classification of objects, which makes many problems easier to model. It also has much better performance than copy semantics, since no memory is copied when creating PCCs. As such, even though we experimented with both, the current implementation of PCCs supports only reference semantics.

In multi-threaded applications, the reference semantics suffers from the general problems associated with the lack of isolation. In programs where changes to the reclassified objects do not need to be reflected back to the originals, copy semantics might be preferred and that is a drawback of our choice to support only reference semantics.

Copy semantics can be achieved with an additional layer on top of PCCs. For example, in one of our main applications of PCCs involving distributed simulations, we designed and implemented a layer on top of PCCs that serializes and deserializes PCCs, and that merges changes made to PCC objects back with the state of their originals. We call this extra layer a **dataframe**. Dataframes could also be implemented in a non-distributed setting, but, so far, we have no strong justification for doing so, since all non-distributed implementations of PCCs can be done much more easily with reference semantics. While interesting in its own right, especially with respect to merging changes, the dataframe layer is not part of PCCs, and hence it is not described here.

3.3 Constant Consistency vs. One-Time Consistency

As the example shows, PCC instances can, at points, be in violation of the predicate that classified them and placed them in the resulting collection. In this example, when active cars reach the final point, their Velocity is set to None (Listing 1, line 21). This, however, does not remove them from the collection, or from the class – ActiveCar – even though from that point on they are in violation of the predicate.

This is a feature, not a flaw, of the design of PCCs. PCCs are designed to be used in iterative computations. As such, at each step, we want stability of the data in the collections. As such, we support **one-time consistency**. The semantics of one-time consistency is as follows: **at the time of creation of a PCC**, before any processing occurs, the state of the instances that make it to the PCC is guaranteed to be consistent with the predicate of the PCC. Once the objects start being processed, however, there are no guarantees of consistency with the predicate that placed them there.

While this choice of semantics may at first seem strange, we note that a similar choice has also been done in mainstream collection libraries. Consider, for example, the iteration over a hash map/dictionary; in most modern programming languages (Java, Python, C#, etc.) the collection cannot be changed during iteration, even if some elements of the collection logically stop belonging to it or new elements logically become members of it. The same happens in sorted collections based on the objects' hashcode, or comparator method: given that the hashcodes and comparisons are

based on the objects' state, when that state changes after the original insertion, the position of the object in the collection may be temporarily inconsistent until explicit adjustments are made.

An alternative to this design choice would be to keep strict constant consistency of the objects with the PCC predicates, and to move objects in and out of collections immediately as their state changes. Constant consistency would make certain instances be reclassified in, and even disappear from, the collection before the processing step would be over. While feasible, this choice would be considerably more complex to implement, slow, and, most importantly, potentially confusing. We decided to keep with the design choices of mainstream collection libraries.

3.4 Inheritance

The relational operations underlying PCCs, of which subsetting is only one, have interesting implications for the traditional concept of inheritance in OOP. Even before presenting the other operations, it is important to clarify the relation between PCCs and inheritance; subsetting serves as a good illustration of the subtleties of this relation. More subtleties will emerge when we present other PCCs. The overall design principle is that PCCs can use inheritance, but do not try to reappropriate it. When inheritance is desired, programmers can complement the PCC declaration with the inheritance features provided by the host language.

In the example of Listing 1, the PCC ActiveCar is a subset of Car, but does not inherit from the class Car. Because there is no inheritance relation, ActiveCar is not a subtype of Car, nor does it inherit any method from Car. The only elements reused from Car by ActiveCar are the fields explicitly marked as dimensions, in this case ID, Position, and Velocity. The lack of implicit inheritance is intentional: it may very well be that programmers want completely different behavior for ActiveCar objects than that present in Car. When behavioral subtyping is desired, the programmer can add it in the usual manner. In the example, if subtyping had been desired, we would have declared ActiveCar as:

```
@subset(Car)
class ActiveCar(Car):
...
```

In this case, ActiveCar would be a subclass of Car, and all methods and fields would be inherited.

4 Relational Operations of Predicate Collection Classes

A PCC is both a class of objects and a specification for collection of objects of that class. As such, it is defined by both a dimensions rule Γ (Gamma) and an extension rule Ψ (Psi). The predicate collections result from relational operations on collections of existing objects. This section revisits subsets and presents the other main relational operations: projection, cross product (join), union, intersection, and parameterization of collections. A summary of these rules is presented in Tables 2 and 3. This section contains the explanation of these rules.

	Γ **(Fields Rule)**
Subset $\text{Sub}_P(A)$	$\gamma(Sub_P(A)) \supseteq \gamma(A)$
Projection $\text{Proj}(A : f_1, ..., f_m)$	$\gamma(Proj(A : f_1, ..., f_m)) \supseteq \{f_1, ..., f_m\}$
Join $A_1 \times_P A_2$	$\gamma(A_1 \times_P A_2) \supseteq \gamma(A_1) \cup \gamma(A_2)$
Union $A_1 \cup A_2$	$\gamma(A_1 \cup A_2) \supseteq \gamma(A_1) \cap \gamma(A_2)$
Intersection $A_1 \cap A_2$	$\gamma(A_1 \cap A_2) \supseteq \gamma(A_1) \cap \gamma(A_2)$

Table 2 – Dimensions rules (Γ) for PCCs, which dictate the fields of PCCs.

	Ψ **(Extension Rule)**
Subset $\text{Sub}_P(A)$	$\psi(Sub_P(A)) = \{a \in \psi(A) \mid P(a)\}$
Projection $\text{Proj}(A : f_1, ..., f_n)$	$\psi(Proj(A : f_1, ..., f_m)) = \psi(A)$
Join $A_1 \times_P A_2$	$\psi(A_1 \times_P A_2) = \{(a_1, a_2) \in \psi(A_1) \times \psi(A_2) \mid P(a_1, a_2)\}$
Union $A_1 \cup A_2$	$\psi(A_1 \cup A_2) = \{a \in \psi(A_1) \cup \psi(A_2)\}$
Intersection $A_1 \cap A_2$	$\psi(A_1 \cap A_2) = \{a \in \psi(A_1) \cap \psi(A_2)\}$

Table 3 – Extension rules (Ψ) for PCCs, which define which instances are in the PCC.

4.1 Subset

Listing 1 presented an example of a subset PCC. More formally, given a collection A of objects of class A, a subset PCC of A for a certain predicate P, written $Sub_P(A)$, which is also a class, is ruled by the following:

Γ **Rule:** $Sub_P(A)$, as a class, includes all dimensions of class A, and can include additional dimensions of its own.
Ψ **Rule:** $Sub_P(A)$, as a set, consists of instances of class $Sub_P(A)$ constructed from instances in A that honor the given predicate P.

4.2 Projection

Given a collection A of objects of class A, a projection PCC of A over a subset of dimensions of A, written $Proj(A : f_1...f_m)$, is defined by the following:

Γ **Rule:** $Proj(A : f_1...f_m)$, as a class, includes the dimensions of class A onto which it is being projected, $f_1...f_m$, and can include additional dimensions of its own.
Ψ **Rule:** $Proj(A : f_1...f_m)$, as a set, consists of instances of class $Proj(A : f_1...f_m)$ constructed from all instances in A.

Listing 2 shows an example where a class Person, with multiple dimensions, is projected as PCC PersonInfo, in which the objects have only two of the dimensions of

```
1  class Person:
2    @dimension(int)
3    def ID(self): return self._ID
4    @dimension(str)
5    def Name(self): return self._Name
6    # plus 20 other dimensions
7
8  @projection(Person, Person.ID, Person.Name)
9  class PersonInfo:
10   def PrintSummary(self):
11     print ‘‘ID=’’ + str(self.ID) + ‘‘ Name=’’ + self.Name
12
13   @Name.setter
14   def Name(self, value): self._Name = value
15   # More fields and methods for PersonInfo
```

Listing 2 – Projection

```
1  @join(Person, Card, Transaction)
2  class RedAlert:
3    def ___init___(self, p, c, t):
4      self.p = p
5      self.c = c
6      self.t = t
7
8    @staticmethod
9    def ___predicate___(p, c, t):
10       p.id==c.owner and t.card==c.id and
11       t.amount > 2000 and t.holdstate==False
12
13   # Dimensions of Person, Card and Transaction are available. For example:
14   def Protect(self):
15     self.c.holdstate = True
```

Listing 3 – Cross Product (Join)

Person (line 8). PersonInfo can be tasked with methods not available to the objects of the class Person. For example, PersonInfo can have exclusive set access to Name (line 13). This means that any function that uses Person will not be able to change the Name, until the projection PersonInfo is used. By making this explicit we are enforcing a certain protocol for data access that is much more expressive than simple accessibility qualifiers.

4.3 Cross Product (Join)

Given a collection A of objects of class A, and a collection B of objects of class B, the join PCC of A and B under a certain given predicate P, written $(A \times_P B)$, is ruled by the following:

```
1  class Fruit:
2    @dimension(float)
3    def size(self): return self._size
4    ...
5
6  class Lemon(Fruit): ...
7  class Orange(Fruit): ...
8  class Banana(Fruit) ...
9
10 @union(Lemon, Orange)
11 class Citrus(Fruit):
12   def MakeJuice(): return size/50
13   # more fields and methods
```

Listing 4 – Union

Γ **Rule:** $A \times_P B$, as a class, includes all dimensions of class A and all dimensions of class B, either directly as the same dimensions or possibly indirectly via instances of A and B as fields, and can include additional dimensions of its own. Ψ **Rule:** $A \times_P B$, as a set, consists of instances of class $A \times B$ constructed from instance pairs $(a \in A, b \in B)$ that honor the given predicate P.

The cross product of two sets is defined as the set of all combinations of elements of the first set and elements of the second, possibly with some additional constraints. In Listing 3, the declaration **@join(Person, Card, Transaction)** establishes that RedAlert is a cross product operation between instances of those three classes. The predicate P, in this case, establishes some constraints regarding identifiers, amounts and status of the transactions.

4.4 Union

Given a collection A of objects of class A, and a collection B of objects of class B, the union PCC of A and B, written $(A \cup B)$, which is also a class, is ruled by the following:

Γ **Rule:** $A \cup B$, as a class, includes all dimensions resulting from the intersection of the dimensions of class A and class B, and can include additional dimensions of its own. Ψ **Rule:** $A \cup B$, as a set, consists of instances of class $A \cup B$ constructed from all instances $a \in A$ and $b \in B$.

While it is possible to define the union of two sets with elements of the same type (e.g. List<Car> c1 ∪ List<Car> c2), it is more interesting to add sets with elements of different types, producing a union that can have meaningful semantic differences from the original sets. Listing 4 shows one such example, where the PCC Citrus defines a set containing all instances of Lemon and Orange, but not of Banana.

As the example illustrates, Unions are an expression a mechanism that can be seen as post-hoc inheritance, i.e. commonalities that are established later rather being modeled in. In the example of Listing 4, using traditional inheritance, a Citrus class

would be defined as inheriting from Fruit, and then classes Lemon and Orange would inherit from Citrus. With the union PCC, such strict hierarchies are not necessary, as new unions can be added relating existing classes to each other externally to their inheritance definitions.

Dimension compatibility in unions is determined by their names. The less fields the elements have in common, the less fields will be available for the methods of the union PCC to use. In the extreme, when the only thing in common between the elements is that they are objects, no fields are available in the union PCC; but the union is still valid.[2]

Because PCCs relate to their domain sets structurally, the union of sets applies to any sets, not just those whose elements have a common user-defined super type, as in the example. The following code shows the union of two sets that have elements with no type in common, other than Object, but that have one field in common, size:

```
class Car:
  @dimension(float)
  def size(self): return self._size

class Lemon:
  @dimension(float)
  def size(self): return self._size

@union(Lemon, Car)
class Prize:
  def Box(): return size + 10
```

In cases where the two sets have some overlap, the union, by default, will include only one of the duplicate objects (so, the DISTINCT semantics of union in SQL). For example:

```
class Car: ...
@subset(Car)
class ActiveCar(Car): ...
@subset(Car)
class RedCar(Car):...

@union(ActiveCar, RedCar)
class RedOrActive(Car): ...
```

In this example, there may be cars that are both active and red. By default, those objects appear only once in the resulting collection RedOrActive.

4.5 Intersection

Given a collection A of objects of class A, and a collection B of objects of class B, the intersection PCC of A and B, written $(A \cap B)$, which is also a class, is ruled by the following:

[2]Many SQL engines support attribute renaming in order to enrich the expression of unions and intersections. For the time being, our model and language does not support field renaming, although it can easily be added in the future.

```
1  class Car: ...
2
3  @subset(Car)
4  class ActiveCar(Car): ...
5
6  @subset(Car)
7  class RedCar(Car): ...
8
9  @intersection(ActiveCar, RedCar)
10 class RedActiveCar(Car): ...
```

Listing 5 – Intersection

> Γ **Rule:** $A \cap B$, as a class, includes all dimensions resulting from the intersection of the dimensions of class A and class B, and can include additional dimensions of its own. (Similar to union)
> Ψ **Rule:** $A \cap B$, as a set, consists of instances of class PCC_I constructed from all instances that belong to the intersection of A and B.

As stated above, the Γ rule is the same as for unions. This requires some explanation, as it may be surprising. Unions and intersections must operate on structurally compatible objects, hence the intersection of the fields in both cases, which gives the minimum common structure. The differences are in the extension (Ψ) rule, i.e. the objects that end up in the resulting set. Our design decision for unions and intersections follows that of SQL's unions and intersections.

Listing 5 shows an example where the set of active cars is intersected with the set of red cars, resulting in a set of cars that are both active and red.

Although the concept of an object being an instance of ActiveCar and of RedCar simultaneously may seem strange, this is explained by the fact that PCCs create objects that are entangled with their originals: a specific car that is both active and red will end up having an incarnation as ActiveCar and another as RedCar; but, due to entanglement, object identification is never lost. Therefore, two PCC instances derived from the same original object pass the equality test.

4.6 Parameterized Collections

Additionally to the basic algebraic operations on collections, we also support parameterization of queries when constructing PCCs. Listing 6 shows one example in the domain of graphs (nodes and edges). The PCC collection InEdge is defined as a parameterized subset. InEdge is a subset of Edge. However, the subset cannot be instantiated without providing specific run-time context: the Node to which the subset collection of Edges are incident upon. Node is therefore the parameter. A Node has to be passed during the creation of the InEdge collection for it to be successful.

Parameters can be also collections of objects. Listing 7 illustrates this with the concept of a pedestrian in danger of being hit by a car. The parameter is a collection (list) of cars.

```
class Node(object):
  @dimension(int)
  def id(self): return self._id
  ...
class Edge(object):
  @dimension(Node)
  def start(self): return self._start
  @dimension(Node)
  def end(self): return self._end
  ...

@parameter(Node, mode=Singleton)
@subset(Edge)
class InEdge(Edge):
  @staticmethod
  def ___predicate___(e, n): return e.end.id == n.id
```

Listing 6 – Parameterized subset with single parameter

```
@parameter(Car, mode=Collection)
@subset(Pedestrian)
class PedestrianInDanger(Pedestrian):
  @staticmethod
  def ___predicate___(p, cars):
    for c in cars:
      if abs(c.Position.X − p.X) < 130 and c.Position.Y == p.Y:
          return True
    return False
```

Listing 7 – Parameterized subset with a list as parameter

4.7 Final Note on PCC Creation

This section showed the different relational operations that underlie the different kinds of PCCs, but it didn't yet cover how these PCCs are constructed from existing collections of objects. As briefly mentioned in Section 2, PCCs are created with the `pcc.create` function – see Listing 1, lines 23–27. This function takes a PCC name (the class to be constructed), one or more lists of objects from which to select the members of the PCC, and possibly additional parameters. For completeness' sake, the following shows examples of how to obtain the different PCCs explained in this section.

Type	**Listing**	**Creation example**
Subset	1	cars = *list of cars* acars = pcc.create(ActiveCar, cars)
Projection	2	persons = *list of persons* pinfos = pcc.create(PersonInfo, persons)
Join	3	persons, cards, trans = *lists of persons, cards, and transactions* ralert = pcc.create(RedAlert, persons, cards, trans)
Union	4	fruits = *list of several types of fruits* citrus = pcc.create(Citrus, fruits)
Intersection	5	cars = *list of cars* acars = pcc.create(ActiveCar, cars) rcars = pcc.create(RedCar, cars) racars = cc.create(RedActiveCar, acars, rcars)
Parameterization	6	edges = *list of edges*; node = Node() inedges = pcc.create(InEdge, edges, params=(node,))
Parameterization	7	peds = *list of pedestrians*; cars = *list of cars* pdanger = pcc.create(PedestrianInDanger, peds, params=(cars,))

5 Usage Examples

We have used PCC in both small algorithms and as a component of a simulation framework that we are developing. This section illustrates the expressiveness of PCCs using four of those examples. All of these examples are available from the PCC repository in Github (See `https://github.com/Mondego/pcc`). Parts of the code are omitted, so that the code shown here can fit in one page. The indentation shown here deviates from the stylistic indentation of Python code, for the same reason. Please see the project repository for the complete code.

5.1 K-Nearest Neighbor

Listings 8 and 9 give the PCC implementation of the K Nearest Neighbor algorithm as applied to the clustering of flowers.[3] The goal here is to identify the type of a flower using four characteristics of flowers: the width and length of the sepal and petals. The example code is shown in two parts, the data model (Listing 8) and the procedural part that uses it (Listing 9).

The main class in this example is `flower` (Listing 8, lines 1–15). A parameterized subset `knn` is defined (Listing 8, lines 17–36), whose purpose is to model the K nearest neighbors of any given flower. As such, it requires two parameters: a flower, and the

[3] Adapted from `http://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/`.

```
1  class flower(object):
2      @dimension(float)
3      def sepal_length(self): return self._sepal_length
4      @dimension(float)
5      def sepal_width(self): return self._sepal_width
6      @dimension(float)
7      def petal_length(self): return self._petal_length
8      @dimension(float)
9      def petal_width(self): return self._petal_width
10     @dimension(str)
11     def fl_type(self): return self._fl_type
12     @dimension(str)
13     def predicted_type(self): return self._predicted_type
14     def ___init___(self, sl, sw, pl, pw, tp):
15         # initialize fields
16
17 @parameter(flower, int)
18 @subset(flower)
19 class knn(object):
20     @staticmethod
21     def euclideanDistance(fl1, fl2):
22         return math.sqrt(pow((fl1.sepal_length - fl2.sepal_length), 2)
23                          + pow((fl1.sepal_width - fl2.sepal_width), 2)
24                          + pow((fl1.petal_length - fl2.petal_length), 2)
25                          + pow((fl1.petal_width - fl2.petal_width), 2))
26     @staticmethod
27     def ___query___(training_flowers, test, k):
28         final_items = sorted([tr_f for tr_f in training_flowers],
29                              key = lambda x: knn.euclideanDistance(test, x))
30         return [final_items[i]
31                 for i in range(len(final_items))
32                 if knn.___predicate___(i, k)]
33     @staticmethod
34     def ___predicate___(index, k): return index < k
35
36     def vote(self): return self.fl_type
```

Listing 8 – K Nearest Neighbor – data model

```
1  def getResponse(knns):
2      classVotes = {}
3      for one_neighbour in knns:
4          response = one_neighbour.vote()
5          classVotes[response] = classVotes.setdefault(response, 0) + 1
6      sortedVotes = sorted(classVotes.iteritems(), key = lambda x: x[1], reverse = True)
7      return sortedVotes[0][0]
8
9  def main():
10     trainingSet, testSet, predictions = [], [], []
11     split, k = 0.67, 3
12     loadDataset("iris.data", split, trainingSet, testSet)
13     for one_flower in testSet:
14         with pcc.create(knn, trainingSet, params = (one_flower, k)) as knns:
15             one_flower.predicted_type = getResponse(knns)
16     print('Accuracy: ' + repr(getAccuracy(testSet)) + '%')
17
18 main()
```

Listing 9 – K Nearest Neighbor – algorithm

number of nearest neighbors to be selected. A few facts are interesting about the `knn` PCC:

- The `knn` PCC does not inherit from flower, but it is a subset of flower collections. As such, all dimensions of the `flower` class are available in `knn` instances. Methods, however, are not inherited.
- The ___query___ defined in the `knn` PCC is tasked with sorting the training set of flowers using the Euclidean distance with the characteristics of the given flower as dimensions (Listing 8, lines 21–25). It selects and returns the nearest K flowers. The default ___query___ cannot be used to create this subset, because the final subset is a limited collection that depends on sorting order. Additional controls like ___order_by___ and ___limit___ would be needed in order to provide this functionality, something we still do not support. Advanced queries such as this one can be directly defined by the programmer.
- The `knn` PCC defines an additional method `vote` (Listing 8, line 36), which returns the label of the flower as neighbor of some other flower. In other words, voting is not a general behavior of flowers; it's only a behavior of flowers that are K-distance similar to some other flower.

Listing 9 shows the procedural part. After loading the data and dividing it into a training and test set (Listing 9, line 20), we iterate over every flower that has to be labeled (Listing 9, lines 21–23). For each of those flowers, we create the corresponding `knn` subset with the flower to be labeled and the number of neighbors to look for as the parameters (Listing 9, line 22). We then decide what type of flower this is by polling its nearest neighbors for votes (Listing 9, line 23). In the `getResponses` function, each of the neighbors votes on a label (line 8), and the majority wins (lines 11-12).

5.2 Car and Pedestrian

We also used PCCs to develop proof-of-concept simulations. The simulation example we worked on involved cars and pedestrians moving towards each other. When a pedestrian is in danger of colliding with a car, it gets out of the way. In this simulation, PCCs were used to drive the state changes for cars and pedestrians, rather than being used to calculate intermediate steps like in the previous example. PCCs are both creators and consumers of the state change.

Listing 10 shows the base classes that were used for this example. Car and Pedestrians have their own classes (lines 1–17 and lines 19–45 respectively). Both classes are derived from the Sprite class in pygame, a module that was used to render the car and pedestrians in the simulation viewer. The attributes required for visualization are not registered as dimensions, and so they will not be copied when PCCs are created from Car and Pedestrian.

Listing 11 details the mechanism needed to move a car. We track two states: InactiveCar (defined in lines 1–9) and ActiveCar (defined in lines 11–21). An InactiveCar is one that has its Velocity dimension set to zero. ActiveCar is naturally the opposite. Both are Subsets of Car, but they do not inherit from Car. This means that they get the dimensions from Car (as they are declared as Subsets) but they do not inherit the methods or the non dimension attributes from Car. This is useful in this case as Car is also a sprite object and has properties and methods that are not needed.

ActiveCar and InactiveCar also define their own methods relevant to the state of the car that it models. An ActiveCar cannot be started, and an InactiveCar cannot be Moved. When an InactiveCar is started, in the next iteration it gets classified as an ActiveCar and gains the ability to move. This is done by two parallel threads, one that starts InActiveCars (lines 26–35) and one that moves ActiveCars (lines 37–44). All the pcc changes are done under the scope of the dataframe object. The dataframe object performs two tasks. First, it uses a synchronization lock to make the computation within its scope thread safe. Second, it allows us to create PCC objects using the copy semantics, and provides a mechanism to copy changes performed on the copied object back to the original. This merge operation is performed at the end of its scope.

Listing 12 shows us the PCC classes, and the mechanisms needed to move pedestrians, and make them avoid danger when they are close to cars. StoppedPedestrian (lines 1–6), and Walker (lines 8–13) are similar to ActiveCar, and InactiveCar classes. They are subsets of Pedestrian that track the state of the pedestrian object. A StoppedPedestrian is one that has not moved from the initial position. A Walker is a pedestrian that has moved from the initial position. There are two threads that create and use these objects in the same way as cars. To make the walkers avoid cars, the PedestrianInDanger class (lines 15–28) was created. This is a parameterized subset of Pedestrians, that takes a list of Cars as a parameter. To shorten the search space, it can also be made a subset of Walker, and parameterized on a list of ActiveCar. The Move function in the PedestrianInDanger class overrides the Move in the Pedestrian class. So when an object gets classified as a PedestrianInDanger, the Move that is executed is different, and it will avoid the car (lines 44–47).

There are many ways in which PCCs allow pedestrians in danger to be calculated. Appendix A.1 shows a couple of different ways of modeling the pedestrians in danger situation.

```
1  class Car(pygame.sprite.Sprite):
2      # The class that shows the image of a car.
3      # Normal class that holds the state of a car.
4      FINAL_POSITION = 500
5      SPEED = 10
6      @dimension(str)
7      def ID(self): return self._ID
8      @dimension(tuple)
9      def position(self): return self._position
10     @dimension(tuple)
11     def velocity(self): return self._velocity
12
13     def ___init___(self, position):
14         # Constructor
15
16     def update(self):
17         # Changes the position of the car in the graphics window.
18
19 class Pedestrian(pygame.sprite.Sprite):
20 # base pedestrian class
21     INITIAL_POSITION = (400, 0)
22     SPEED = 10
23     @dimension(str)
24     def ID(self): return self._ID
25     @dimension(int)
26     def X(self): return self._X
27     @dimension(int)
28     def Y(self): return self._Y
29
30     def ___init___(self):
31         # Constructor
32
33     def Move(self):
34         self.X -= Pedestrian.SPEED
35         if self.X <= 0:
36             self.Stop()
37
38     def Stop(self):
39         self.X, self.Y = Pedestrian.INITIAL_POSITION
40
41     def SetPosition(self, x):
42         self.X = x
43
44     def update(self):
45         # updates the graphics with changes to state.
```

Listing 10 – Normal classes needed for the Car and Pedestrian Simulation

```
1  @subset(Car)
2  class InactiveCar(object):
3      # Car that is not moving, Velocity is zero
4      @staticmethod
5      def ___predicate___(c):
6          return c.velocity == (0, 0, 0) or c.velocity == None
7
8      def Start(self):
9          self.velocity = (Car.SPEED, 0, 0)
10
11 @subset(Car)
12 class ActiveCar(object):
13     # car that is moving, velocity is not zero
14     @staticmethod
15     def ___predicate___(c):
16         return not (c.velocity == (0, 0, 0) or c.velocity == None)
17
18     def Move(self):
19         x,y,z = self.position
20         xvel, yvel, zvel = self.velocity
21         self.position = (x + xvel, y + yvel, z + zvel)
22
23     def Stop(self):
24         self.position, self.velocity = (0,0,0), (0,0,0)
25
26 def StartInactiveCars(cars, MainWindow):
27     # Starts inactive cars every 5 secs
28     while True:
29         with dataframe(carlock) as df:
30             iacs = df.add(InactiveCar, cars)
31             for car in iacs:
32                 car.Start()
33                 register(car.ID, cars, MainWindow)
34                 break
35         sleep(5)
36
37 def MoveActiveCars(cars, MainWindow):
38     # Moves active cars every 300 ms
39     while True:
40         with dataframe(carlock) as df:
41             acs = df.add(ActiveCar, cars)
42             for car in acs:
43                 car.Move()
44         sleep(0.3)
```

Listing 11 – PCC classes and its usage needed to move the Car

```
1  @subset(Pedestrian)
2  class StoppedPedestrian(Pedestrian):
3      # A person that is not moving
4      @staticmethod
5      def ___predicate___(p): return p.X, p.Y == Pedestrian.INITIAL_POSITION
6
7  @subset(Pedestrian)
8  class Walker(Pedestrian):
9      # A person who is walking.
10     @staticmethod
11     def ___predicate___(p): return p.X, p.Y != Pedestrian.INITIAL_POSITION
12
13 @parameter(list)
14 @subset(Pedestrian)
15 class PedestrianInDanger(Pedestrian):
16     # A person who is in danger of colliding with a car
17     @staticmethod
18     def ___predicate___(p, cars):
19         for c in cars:
20             cx, cy, cz = c.position
21             if cy == p.Y and abs(cx - p.X) < 70:
22                 return True
23         return False
24
25     def Move(self):
26         self.Y += 50
27
28 def StartPedestrian(peds, MainWindow):
29     # Make a stopped pedestrian walk every 3 secs.
30
31 def MovePedestrian(peds, cars, MainWindow):
32     # Make a Walker move.
33     while True:
34         with dataframe(pedlock) as df:
35             pids = df.add(PedestrianInDanger, peds, params = (cars,))
36             wks = df.add(Walker, peds)
37             for p in (pids + wks):
38                 pid.Move()
39         _sleep(0.5)
```

Listing 12 – PCC classes and its usage needed to move Pedestrians

Figure 3 – Urban simulation.

5.3 Distributed Simulations: The Spacetime Framework

The previous example is a proof-of-concept of a much more complex framework that we are developing for collaborative distributed simulations, called the Spacetime Framework (`https://github.com/Mondego/spacetime`). The details of the Spacetime Framework are out of the scope of this paper, but we present a short summary. This is, by far, our most meaningful application of PCCs.

5.3.1 Background: Modeling and Simulation

Modeling and simulation is a mature field in both research and development. When studying some real-world process or activity, one starts with developing a model for it, typically a mathematical model of some kind that abstracts away unnecessary complexity; then, when the model does not have a closed form solution, computer simulation can be used to study its characteristics (behavioral and performance). This approach to studying the real world is used in almost all branches of Science and Engineering.

Microscopic model simulations can capture richer real world scenarios than macroscopic ones, but they require much greater computational resources. The finer the level of granularity and the larger the number of individual units modeled, the more resources they need. For that reason, these simulations tend to be limited in size and/or purpose. There is no easy way of designing and implementing multi-purpose simulations; each study requires not just its own separate model, something that would be expected, but also a separate simulation.

Over the past few years, we have been involved in simulation projects that challenge this state of affairs – see picture in Figure 3 showing one of the 3D simulated cities. These projects have evolved towards multi-purpose microscopic simulations of urban areas. It has become clear that there are lines of expertise for the different subsystems, and that, for that reason alone, we need a decentralized simulation architecture allowing different groups to provide relatively independent simulations of their models, but in a coordinated way, because the models are mutually interdependent.

PCCs were designed to address this application domain, and its need to support separation of concerns at the systems design level.

5.3.2 Spacetime Frames: Time-Framed PCCs

Our use of PCCs in time-discrete simulations combines them with time frames, resulting in what we call *spacetime frames.* In time-discrete simulations [Fuj00], each simulation unit consists of a stepping function (*stepper*, for short) that is executed periodically, and that changes the state of the world at each step. Each step is a time frame. Steppers pull data in the form of PCCs from the shared store in the beginning of a

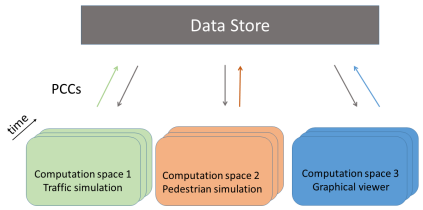

Figure 4 – Architecture of the Spacetime simulation framework.

time frame, and may push data back at the end of each time frame. Hence the concept of spacetime frames: local snapshots of global world state that are used during a time window, and whose changes may be reflected back to the global state at the end of the time frame.

An external data store is shared by several simulation units (see Figure 4). One important part of the use of PCCs in the Spacetime Framework is that the predicates for selecting objects are run at the store, therefore minimizing the amount of data that is brought in over the network. PCCs are critical in allowing different developers to classify raw shared data according to the role that such data plays in their local simulations. For example, the state of cars may be used by one simulation to construct graphical objects, while being used by another one to construct polluting objects.

6 Related Work

Since the early days of OOP, the simple use of classes and single inheritance has been questioned. From multiple inheritance to traits to predicate classes (and dispatch), many alternatives have been proposed over the years. Here we cover some of the work most related to PCCs.

6.1 Predicate Classes

Some of the inspiration for PCCs, including the name, comes from predicate classes [Cha93]. The idea behind predicate classes is for objects to be dynamically classified, taking new/different behavior as they change state. Objects that satisfy predicates specified in predicate classes automatically become instances of those classes; when they stop satisfying those predicates, they stop being instances of those classes. This is very similar to PCCs, but there are some important differences.

The major difference is that PCCs, as the name indicates, pertain to collections of objects, rather than to individual objects. This difference changes the focus and the capabilities of the basic idea substantially. In the case of simple predicate classes, the programmer simply states the predicate to be satisfied (e.g. a car whose velocity is zero), but there are no handles for collections of objects that satisfy those predicates at any point in time. The PCCs' focus on collections not only exposes these handles but also enables the full expressive power of relational operations on collections, such as subsetting, projection, cross product, etc. Simple predicate classes express implicit subsets only: subsets, because predicates on field values constrain the state of the parent objects; implicit, because there is no handle for that subset.

More importantly, one of the goals underlying simple predicate classes is to always ensure the satisfiablity of the predicate. For example, given a normal class buffer and a predicate class empty buffer, if the last element is removed from a buffer object, that object immediately[4] becomes an instance of the empty buffer predicate class; similarly, if an element is added to an empty buffer object, that object immediately stops being an instance of the empty buffer predicate class. Classification is always consistent with the state of the objects. That is not a goal of PCCs. PCCs classify the objects at some point in time, at which point the predicate is guaranteed to be satisfied; but once included in the collection, the state of the objects may later change, possibly becoming inconsistent with the predicate that placed them in the data frame. That is not just acceptable: it is a desired semantics. PCCs are meant to hold a fixed collection of objects whose state can change. Take, for example, the case of a collection of non-empty buffer objects; if during subsequent processing all elements are removed from a given buffer object in that collection, we still want that buffer object to be in the collection, even though it is empty; we don't want it to suddenly disappear because it became empty. So the semantics of predicates in PCCs is quite different from that of predicates in simple predicate classes: always true (in the case of simple predicate classes) vs. true at the time of data frame creation (in the case of PCCs).

The relaxation of satisfiability is also what makes it possible to implement relational operations in practice, not just subsetting. Unlike subsetting, that looks only at internal state of the objects, joins (cross product) and parameterization pertain to combinations of objects. Take for example, a join between cars and their owners. If at some point in the framed computation the car ownership changes from one person to another (or to no one), we would need to search combinations of a car and persons again to check whether the resulting join object satisfies the predicate. Strict satisfiability of predicates for objects involved in join operations, as well as parameterizations, would be prohibitive to implement.

6.2 Other Class-Instance Associations

Besides predicate classes, the OOP literature presents a considerable number of ideas aimed at making the instance-class relationships more flexible. We describe some of them here, and how they relate to PCCs.

Fickle [DDDCG01] includes another idea for dynamic object reclassification that is not based on predicates, but on explicit reclassification by the programmer. The Fickle language provides a construct to reclassify instances of special "state" classes that can be used by programmers. These special classes, however, cannot be used as types for fields of parameters, as that would violate type safety. The main difference between PCCs and this older work is, again, the focus on collections rather than on individual instances. Additionally the Fickle reclassification construct is not declarative but imperative in nature. In contrast, PCCs are defined using declarations (the predicates).

First introduced in Flavors by Moon [Moo86], and then in CLOS, mixins (abstract subclasses) are a powerful way of combining object behavior, as they can be applied to existing superclasses in order to create a related family of modified classes. Bracha and Cook introduced mixin-based inheritance [BC90], a form of class inheritance formulated as a composition of mixins. Mixin layers [SB98] are a form of decomposition that targets the encapsulation of class collaborations: each class encapsulates several roles,

[4] "Immediately" here includes lazyness, i.e. not necessarily instantly but as soon as classification is needed.

where each role embodies a separate aspect of the class's behavior. A cooperating suite of roles is called a collaboration. Mixins are only vaguely related to PCCs in that they make reuse of behavior more flexible than inheritance, allowing objects to be given several different roles. But the classification is still static, meaning that it is established before any instances are created. In contrast, PCCs serve to reclassify objects at runtime.

Self [US87], and many languages inspired by it, including recent ones such as YinYang [ME14], include the concept of dynamic inheritance, which "allows the inheritance graph to change during program execution." While PCCs focus on the dynamics of program execution, our goal is not to change the inheritance hierarchy at runtime, but to change the classification of objects among existing (fixed) classes at runtime. In our model, the class hierarchy is static, as it reflects the important activity of designing and modeling the application entities; but the classification of data can change at runtime based on specific predicates on the state of the objects.

Bertino and Guerrini proposed a technique that allows objects to belong simultaneously to multiple [most specific] classes [BG95]. The motivation was data modeling situations in which a single instance (e.g. a person) is naturally associated with multiple classes (e.g. student, and female). Although similar to multiple inheritance, the technique proposed in that paper aimed at avoiding the proliferation of subclasses that are simple combinations of other classes. This work built on the idea of mixin-based inheritance [BC90], and it predates traits [SDNB03, OZ05]. Traits are another way of reusing behavior. PCC instances do not have traits, but rather they are instances associated with a single class, that take the state from existing objects.

Finally, virtual classes [MMP89, EOC06], dependent classes [GMO07], and generics [MMN75, BOSW98] are mechanisms to parameterize classes. That work is vaguely related to PCCs in that it is particularly useful for collection classes such as lists, sets, etc. But the purpose of parameterized classes is quite different from that of predicate [collection] classes: the former targets the generalization of type definitions (types parameterized on other types), whereas the latter targets the association between instances and their classes.

7 Conclusions and Future Work

This paper has introduced the concept of Predicate Collection Classes, PCCs for short. PCCs are a declarative mechanism of selecting objects from collections, reclassifying them along the way. PCCs are both classes and specifications of collections of objects of those classes. Composition of collections can be expressed very easily using concepts from relational algebra such as subsetting, projection, cross product, union and intersection. PCCs are useful for filtering and manipulating collections, including when the elements of those collections may behave differently depending on which collection they are placed.

We have implemented PCCs in Python via Python's decorators. Our implementation is publicly available at `https://github.com/Mondego/pcc`, and can be installed via Python's pip.

We have successfully used PCCs to model several iterative algorithms that use collections heavily. This showed us that PCCs are an expressive mechanism for declaring operations on collections while, at the same time, establishing new behaviors for objects that fall into those collections. Our most serious use of PCCs so far is on a framework we are developing for distributed collaborative simulations called

Spacetime Framework (`https://github.com/Mondego/spacetime`). PCCs are a core component of that framework, as they allow the declarative specification of optimized dataframes that are streamed from a data server to distributed simulation components, allowing each component to give their own behavior to the data – that behavior may change over time. The Spacetime Framework was used in a graduate-level course on distributed simulations; the students' reactions to PCCs so far have been quite positive. The framework is currently being integrated in a product related to urban simulations. While the first impressions are encouraging, in this paper we do not provide any systematic evidence of benefit.

PCCs are inspired by relational query languages, which have proven to follow a timeless model for manipulating collections of data. As future work, we plan to: (1) extend the expressibility of our PCC language to better match SQL, (2) strengthen the pre-runtime checks of the programmer's declarations, (3) further explore the application of PCCs to refactor existing libraries, and (4) assess the tradeoffs on performance vs. elegance when comparing PCCs with non-PCC programs. We also plan to implement PCCs in a statically-typed language, namely C#.

References

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. URL: `http://doi.acm.org/10.1145/97945.97982`, `doi:10.1145/97945.97982`.

[BG95] Elisa Bertino and Giovanna Guerrini. Objects with multiple most specific classes. In *ECOOP'95-Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 102–126. Springer, 1995. `doi:10.1007/3-540-49538-X_6`.

[BKSW09] Bard Bloom, Paul Keyser, Ian Simmonds, and Mark Wegman. Ferret: Programming language support for multiple dynamic classification. *Computer Languages, Systems & Structures*, 35(3):306–321, 2009. `doi:10.1016/j.cl.2008.05.005`.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM. URL: `http://doi.acm.org/10.1145/286936.286957`, `doi:10.1145/286936.286957`.

[Bru02] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.

[Cha93] Craig Chambers. Predicate classes. In *ECOOP'93-Object-Oriented Programming*, pages 268–296. Springer, 1993. `doi:10.1007/3-540-47910-4_15`.

[Chr76] Nicos Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[DDDCG01] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object reclassification. In *ECOOP 2001-Object-Oriented Programming*, pages 130–149. Springer, 2001. doi:10.1007/3-540-45337-7_8.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1111037.1111062, doi:10.1145/1111037.1111062.

[Fuj00] Richard M. Fujimoto. *Parallel and Distributed Simulation*. Wiley, 2000. doi:10.1002/9780470172445.ch1.

[GMO07] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 133–152, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1297027.1297038, doi:10.1145/1297027.1297038.

[Lev79] Azriel Levy. *Basic Set Theory*. Springer-Verlag, 1979. doi:10.1007/978-3-662-02308-2_1.

[ME14] Sean McDirmid and Jonathan Edwards. Programming with managed time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 1–10, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2661136.2661145, doi:10.1145/2661136.2661145.

[MMN75] R. Milner, L. Morris, and M. Newey. *A Logic for Computable Functions with Reflexive and Polymorphic Types*, pages 371–394. IRIA-Laboria, 1975.

[MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM. URL: http://doi.acm.org/10.1145/74877.74919, doi:10.1145/74877.74919.

[Moo86] David A. Moon. Object-oriented programming with flavors. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 1–8, New York, NY, USA, 1986. ACM. URL: http://doi.acm.org/10.1145/28697.28698, doi:10.1145/28697.28698.

[Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Confer-*

ence on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1094811.1094815, doi:10.1145/1094811.1094815.

[SB98] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 550–570, London, UK, UK, 1998. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646155.679703, doi:10.1007/bfb0054107.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. *ECOOP 2003 – Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings*, chapter Traits: Composable Units of Behaviour, pages 248–274. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. URL: http://dx.doi.org/10.1007/978-3-540-45070-2_12, doi:10.1007/978-3-540-45070-2_12.

[Spa] Apache spark. http://spark.apache.org/. doi:10.1002/9781119183464.ch11.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. URL: http://doi.acm.org/10.1145/38765.38828, doi:10.1145/38765.38828.

A Additional Examples

A.1 Car and Pedestrian: Alternative Models

Section 5.2 showed the proof-of-concept simulation example with cars and pedestrians, including the concept of PedestrianInDanger. There are several ways of modeling the situation where a pedestrian is in danger of colliding with a car. Here we show two additional ones.

Listing 13 shows the definition of a join class CarAndPedestrianNearBy. This class is a join of cars, and pedestrians that meet the condition of the predicate. When executed, a collection of pairs of cars and pedestrians will be created. However, if the wish is to collect an object representing one pedestrian and all cars that it is in danger of colliding with, the PedestrianAndManyCarsNearby class defined in Listing 14 can be used. A collection of cars is stored in the dimension cars instead of just one car as seen in the previous alternative. The form of the dimensions in the join class is up to the developers needs and creativity. Object references found within the join object are tracked.

A.2 Page Rank

Listing 15 shows an implementation of the PageRank algorithm using PCCs. There are two main classes: Node (lines 1–6) and Edge (lines 8–13). Edge objects refer to two nodes, the start node (line 10) and the end node (line 12). InEdge (lines 15–19) and OutEdge (lines 21–25) are subsets of Edge sets parameterized on a node, so it models

```
1  @join(Car, Pedestrian)
2  class CarAndPedestrianNearBy(object):
3      @dimension(Car)
4      def car(self): return self._car
5      @dimesnion(Pedestrian)
6      def pedestrian(self): return self._pedestrian
7
8      def ___init___(self, car, pedestrian):
9          self.car, self.pedestrian = car, pedestrian
10
11     @staticmethod
12     def ___predicate___(p, c):
13         cx, cy, cz = c.position
14         if cy == p.Y and abs(cx − p.X) < 70:
15             return True
16         return False
17
18     def Move(self):
19         self.pedestrian.Y += 50
```

Listing 13 – An alternative to PedestrianInDanger using joins

the in-edges and out-edges, respectively, of any given node. To calculate PageRank for each node (line 34–onward), we iterate several times through the list of nodes until the amount of change crosses a certain threshold (line 34). In each iteration, and for each node, the collection of the in-edges of the node is created (line 39). Then, for each in-edge, we compute the out-edges of the start node (line 41), and accumulate the normalized count (line 42). Finally, we update the PageRank value (line 43).

The main take away from this simple example is that the logic of finding in-edges and out-edges of nodes is **declared** in the form of PCCs, instead of being imperatively embedded in the algorithm. Modeling important set operations as classes makes the algorithm more readable. The PCC programming style forces programmers to make deliberate changes to the predicate of the PCCs in order to make changes to the logic of the program.

A.3 Travel Salesman Problem

The traveling salesman problem is an optimization problem, where, given a finite number of cities and knowing the cost of travel between these cities, an optimal route of travel must be calculated for a salesman with the sole condition that the salesman does not visit the same city more than once. This can be modeled as a undirected weighted graph, with cities as vertices, and the cost of travel between cities as the weights of the edge between the corresponding vertices. The Christofides algorithm is one of many approximate solutions for this problem [Chr76]. Listings 16, 18, 19, 20 and 17 together show an implementation of this algorithm using PCCs.

Listing 16 shows the definitions of the main classes in the example: City (lines 1–9) and Path (lines 11–23). Every City has a name dimension and a dimension to determine if it has been connected by a chosen path to another City. Path objects have references to both the cities that they connect, a dimension (distance) that provides the cost of travel between the two cities, and another dimension that determines if it

```
1  @join(Car, Pedestrian)
2  class PedestrianAndManyCarsNearby(object):
3      @dimension(list)
4      def cars(self): return self._cars
5      @dimesnion(Pedestrian)
6      def pedestrian(self): return self._pedestrian
7
8      def ___init___(self, cars, pedestrian):
9          self.car, self.pedestrian = cars, pedestrian
10
11     @staticmethod
12     def ___query___(pedestrians, cars):
13         return [(p, [c for c in cars if PedestrianAndManyCarsNearby.___predicate___(c, p)]) for p in pedestrians]
14
15     @staticmethod
16     def ___predicate___(p, c):
17         cx, cy, cz = c.position
18         if cy == p.Y and abs(cx − p.X) < 70:
19             return True
20         return False
21
22     def Move(self):
23         self.pedestrian.Y += 50
```

Listing 14 – Another alternative to PedestrianInDanger using joins

has been chosen or not. When objects of both City and Path are created, a path for every city to every other city is created, but not all are chosen for the final solution.

We start from the procedural part of the algorithm, which has six main steps (Listing 17). The first step is to create a minimum spanning tree (MST) from the vertices and edges. Then, a subgraph is induced by using only the vertices that have odd degree in the MST. A minimum weight, perfect matching is found in this subgraph and combined with the MST to form a connected multigraph. Every vertex in this multigraph has an even degree which is an important prerequisite for the next step. An Eulerian trail is formed from this multigraph which is then made into a Hamiltonian circuit by skipping repeated vertices. In this main function, apart from building the MST, there are no iterations, and there are no updates made to the original collections of cities and paths. PCC are used as a means to retrieve data transformed in certain ways, and used as intermediate steps in calculating the final output. They were used as components in a pipeline. We now proceed to explain the PCC model.

To build the MST from these cities (vertices), and paths (edges), we implemented the Prim's algorithm using three PCC classes. Listing 18 provides the details. ConnectedCity (lines 13–17) can create a collection of all cities that are connected by a chosen path. DisconnectedPath (lines 7–11) is a subset of paths that have not been chosen yet. NearbyDisconnectedPath (lines 19–28) is a parameterized subset of DisconnectedPath. The parameter needed to instantiate this class is the collection of ConnectedCity. This class allows us to retrieve a collection of paths, that have not been chosen (subset of DisconenctedPath), and only one of the cities in the path is connected to a chosen path (by using the list of ConnectedCity sent as a parameter). From this list of paths, we choose the path with the least weight, and make it a chosen

```
1  class Node(object):
2      @dimension(int)
3      def id(self): return self._id
4      @dimension(float)
5      def pagerank(self): return self._pagerank
6      def ___init___(self, id, pagerank): self.id, self.pagerank = id, pagerank
7
8  class Edge(object):
9      @dimension(Node)
10     def start(self): return self._start
11         @dimension(Node)
12     def end(self): return self._end
13     def ___init___(self, n1, n2): self.start, self.end = (n1, n2)
14
15 @parameter(Node)
16 @subset(Edge)
17 class InEdge(Edge):
18     @staticmethod
19     def ___predicate___(e, n): return e.end.id == n.id
20
21 @parameter(Node)
22 @subset(Edge)
23 class OutEdge(Edge):
24     @staticmethod
25     def ___predicate___(e, n): return e.start.id == n.id
26
27 def CreateNodesAndEdges():
28     # ... create nodes and edges
29     return nodes, edges
30
31 nodes, edges = CreateNodesAndEdges()
32 largest_change, allowed_delta, damp = 100, 0.001, 0.85
33
34 while largest_change > allowed_delta:
35     largest_change = 0.0
36     for n in nodes:
37         old = n.pagerank
38         sum = 0.0
39         for inedge_of_n in PCC.create(InEdge, edges, params = (n,)):
40             pg_contributor = inedge_of_n.start
41             outedges_of_contrib = PCC.create(OutEdge, edges, params = (pg_contributor,))
42             sum += pg_contributor.pagerank / len(outedges_of_contrib)
43         n.pagerank = ((1.0 - damp) / len(nodes)) + (damp * sum)
44         diff = n.pagerank - old
45         if diff > largest_change:
46             largest_change = diff
47
48 for n in nodes:
49     print n.id, n.pagerank
```

Listing 15 – PageRank

```
1  class City(object):
2      # Class for a city
3      @dimension(str)
4      def name(self): return self._name
5      @dimension(bool)
6      def isconnected(self): return self._isconnected
7
8      def ___init___(self, name):
9                  # initialize fields
10
11 class Path(object):
12     # Class for a connection between cities
13     @dimension(City)
14     def city1(self): return self._city1
15     @dimension(City)
16     def city2(self): return self._city2
17     @dimension(float)
18     def distance(self): return self._distance
19     @dimension(bool)
20     def isconnected(self): return self._isconnected
21
22     def ___init___(self, city1, city2, distance):
23                 # Initialize Fields
```

Listing 16 – Main classes defined for solution to Traveling salesman problem

path. The PCC objects are used directly without a dataframe to manage entanglement. The objects are entangled by reference and changes made to the PCC objects are reflected instantaneously in the original. These steps are repeated until all cities have been connected. The collection of chosen paths is the minimum spanning tree for this graph. ConnectedPath is a subset of Path that can give us a collection of all chosen paths. After the construction of the minimum spanning tree, ConnectedPath gives us a way to retrieve the edges in that tree. The algorithm is defined in the BuildMst function (lines 30–53).

The rest of the steps are not iterative, so PCC collections was used as a way to compute the output of each step and pass it along to the next step. Listing lst:tspmultigraph shows the PCC classes that are required for steps 2-5 that create the multigraph of even vertices required to compute the Eulerian tour in Step 6.

Step 2 of algorithm is to get a subset of vertices in the MST that have odd degree. The subset class CityWithOddDegree (lines 1–16) was used to retrieve this. It is a subset of cities and takes the MST (ConnectedPath subset) as a parameter.

Step 3 is creating a subgraph from all paths not chosen using only the cities with odd degree in the MST. The subset class PathsWithGivenCities (lines 18–30) is used to obtain the paths in that subgraph. It is a parameterized subset of DisconnectedPaths parameterized on the collection of cities that have odd degree.

The next step in the algorithm is to create a minimum weight perfect matching on the paths obtained in the previous step. A subset of the paths created in the previous step must be chosen such that they satisfy two conditions: every city in the subgraph must be chosen only once, the edges chosen must have the least weight possible. The subset class min_weight_perfect_match (lines 32–46) is used to create this collection.

```
1  def CreateRandomGraph(number):
2      # Load/Create cities, and paths
3      return cities, paths
4
5  def PrintTSPPath(cities, paths):
6      # Step 1: Building a Minimun spanning tree using Prim's Algorithm
7      BuildMst(cities, paths)
8      MST = pcc.create(ConnectedPath, paths)
9      # Step 2: Find all cities in the MST that have odd degree (O)
10     O = pcc.create(CityWithOddDegree, cities, params = (MST,))
11     # Step 3: Find the induced subgraph given by the vertices from O
12     not_MST_paths = pcc.create(DisconnectedPath, paths)
13     subgraph = pcc.create(PathsWithGivenCities, not_MST_paths, params = (O,))
14     # Step 4: Find the Minimum weight perfect matching M in the subgraph
15     M = pcc.create(min_weight_perfect_match, subgraph)
16     # Step 5: Combining the edges of M and T to form a connected multigraph H
17     # in which each vertex has even degree
18     H = pcc.create(multigraph, M, MST)
19     # Step 6: Form an Eulerian circuit in H.
20     eulertour = pcc.create(EulerTour, cities, params = (H,))
21     # Step 7: Making the circuit found in previous step into a Hamiltonian
22     # circuit by skipping repeated vertices (shortcutting).
23     finaltour = pcc.create(HamiltonianTour, eulertour)
24     # Printing out the resulting tour.
25     PrintConnections(finaltour)
26
27 cities, paths = CreateRandomGraph(10)
28 PrintTSPPath(cities, paths)
```

Listing 17 – Christofides algorithm for the Traveling Salesman Problem

```
@subset(Path)
class ConnectedPath(Path):
    # Subset of path, if this path between cities is chosen
    @staticmethod
    def ___predicate___(p): return p.isconnected

@subset(Path)
class DisconnectedPath(Path):
    # Subset of path, if this path between the cities is not chosen
    @staticmethod
    def ___predicate___(p): return not p.isconnected

@subset(City)
class ConnectedCity(object):
    # Subset of cities that are connected by some chosen path.
    @staticmethod
    def ___predicate___(c): return c.isconnected

@parameter(list) #connected_cities
@subset(DisconnectedPath.Class())
class NearByDisconnectedPath(Path):
    # Subset of disconnected paths that can connect to one
    # among the connected cities given
    # in the parameter.
    @staticmethod
    def ___predicate___(dis_path, connected_cities):
        set_cities = set([city.name for city in connected_cities])
        return (dis_path.city1.name in set_cities) ^ (dis_path.city2.name in set_cities)

def BuildMst(cities, paths):
    # Treat a random node as the start,
    # Make it a forest with one node.
    randomstart = random.choice(cities)
    randomstart.isconnected = True
    while True:
        # Collect all Nodes that are part of the forest.
        connected_cities = pcc.create(ConnectedCity, cities)
        # If all nodes are part of the forest exit loop.
        if len(connected_cities) == len(cities):
            break
        # Find all paths that are have not been chosen.
        disconnected_paths = pcc.create(DisconnectedPath, paths)
        # Find all the paths that are not chosen
        # but can be connected to the existing forest.
        all_nearby_discon_paths = pcc.create(NearByDisconnectedPath, disconnected_paths,
                                             params = (connected_cities,))
        if len(all_nearby_discon_paths) != 0:
            # Choose the path with the least weight.
            best_choice_path = sorted(all_nearby_discon_paths, key = lambda x: x.distance)[0]
            # Join it to the forest.
            best_choice_path.isconnected = True
            best_choice_path.city1.isconnected = True
            best_choice_path.city2.isconnected = True
```

Listing 18 – PCCs needed for building the Minimum Spanning Tree using Prim's algorithm

```
@parameter(ConnectedPath)
@subset(City)
class CityWithOddDegree(object):
        # Subset of cities that have Odd degree in the Graph passed as a parameter
    @staticmethod
    def ___query___(cities, paths):
        result = []
        for city in cities:
            pcs = [p for p in paths if city.name in set([p.city1.name, p.city2.name])]
            if CityWithOddDegree.___predicate___(pcs):
                result.append(city)
        return result

    @staticmethod
    def ___predicate___(pcs):
        return len(pcs) % 2 == 1

@parameter(CityWithOddDegree)
@subset(DisconnectedPath)
class PathsWithGivenCities(Path):
    # Subclass of Path,
    # Class to find subgraph of given graph using only given vertices
    @staticmethod
    def ___query___(paths, cods):
        cities = set([cod.name for cod in cods])
        return [path for path in paths if PathsWithGivenCities.___predicate___(path, cities)]

    @staticmethod
    def ___predicate___(path, cities):
        return path.city1.name in cities and path.city2.name in cities

@subset(PathsWithGivenCities)
class min_weight_perfect_match(Path):
    # Subset of path, paths that make up the minimun weighted perfect matching edges
    # using the given vertices.
    @staticmethod
    def ___query___(subgraph_paths):
        objs = maxWeightMatching([(int(p.city1.name), int(p.city2.name), 1-p.distance) \
                                  for p in subgraph_paths], True)
        req_paths = [(str(i), str(objs[i])) for i in range(len(objs)) if objs[i] != -1]
        return [path for path in subgraph_paths for req_path in req_paths \
                if min_weight_perfect_match.___predicate___(path, req_path)]

    @staticmethod
    def ___predicate___(path, req_path):
        return path.city1.name == req_path[0] and path.city2.name == req_path[1]

@union(min_weight_perfect_match, ConnectedPath)
class multigraph(Path):
    # A Union of two graphs
    pass
```

Listing 19 – PCCs needed for building the multigraph with even degree vertices

It is a subset of PathsWithGivenCities, and retrieves the perfect matching.

To create the multigraph in the Step 5, a Union class multigraph (line 48–51) was created using the perfect matching paths from the previous step and the whole MST computed earlier.

Listing 20 shows the PCC classes needed for generating the Hamiltonian tour of the multigraph. First a Eulerian tour of this multigraph is computed using the parameterized subset class EulerTour (line 1–29). It is a subset of City and takes the multigraph as a parameter. The algorithm to find the Eulerian tour was the Hierholzer's algorithm. Since the basic algorithm will sometimes not cover all cities, cities was a parameter that was used to calculate the route from multiple start points and we picked the first route that included all cities. The last step is to make this a Hamiltonian circuit by removing the repeated cities in the loop. This was done using the HamiltonianTour subset class (line 31–47). It is a subset of the EulerTour, and returns unique cities in the order they were found in the EulerTour. This ordered collection of cities is the approximate solution to the traveling salesman problem.

About the authors

Cristina Videira Lopes is a professor in the Bren School of Information and Computer Sciences at the University of California, Irvine. When she's not busy hacking code, she teaches programming and software engineering. She also manages her large research group and is Associate Director of the Institute for Software Research at UC Irvine. Contact her at `lopes@uci.edu`, or visit `http://mondego.ics.uci.edu`.

Rohan Achar is a Ph.D. student in the Bren School of Information and Computer Sciences at the University of California, Irvine. His research interests are programming languages and software engineering.

Arthur Valadares is a Ph.D. candidate in the Bren School of Information and Computer Sciences at the University of California, Irvine. He is developing a simulation framework for distributed collaborative simulations.
`http://www.ics.uci.edu/~avaladar/`.

Acknowledgments C. Lopes would like to thank the members of the IFIP Working Group on Language Design (WG 2.16) who attended the meeting in January 2016, for their useful feedback about the general idea of PCCs.

```
1  @parameter(multigraph)
2  @subset(City)
3  class EulerTour(object):
4      # A subset of cities in order of the Euler Tour Route.
5      @staticmethod
6      def ___query___(cities, paths):
7          for start_city in cities:
8              totake = {}
9              travel = []
10             visited = set()
11             for path in paths:
12                 totake.setdefault(path.city1, set()).add(path)
13                 totake.setdefault(path.city2, set()).add(path)
14             city = [key for key in totake.keys() if key.name == start_city.name][0]
15             visited.add(city.name)
16             while len(totake[city]) > 0:
17                 next_path = totake[city].pop()
18                 travel.append(city)
19                 city = next_path.city1 if city == next_path.city2 else next_path.city2
20                 visited.add(city.name)
21                 totake[city].remove(next_path)
22             travel.append(city)
23             if len(set(travel)) == len(cities):
24                 return travel
25         return []
26
27     @staticmethod
28     def ___predicate___():
29         return True
30
31 @subset(EulerTour)
32 class HamiltonianTour(object):
33     # A subset of cities that represents the
34     # complete tour the travelling salesman takes.
35     @staticmethod
36     def ___query___(cities):
37         seen = set()
38         result = []
39         for city in cities:
40             if HamiltonianTour.___predicate___(city, seen):
41                 result.append(city)
42                 seen.add(city)
43         return result
44
45     @staticmethod
46     def ___predicate___(city, seen):
47         return city not in seen
```

Listing 20 – PCC classes needed for building the Hamiltonian circuit