Probabilistic Relational Agent-based Models (PRAMs) Milestone 3 Report

Tomek D. Loboda[*]

February 12, 2019

This report introduces and describes a software package implementing the PRAM representation and algorithm delineated in the two previous reports. This implementation is a pastiche of the implementation presented by Paul Cohen in the Milestone 2 report. Because the package is in an early stage of life, it is still malleable; it will take time for the API (i.e., Application Programming Interface) to crystallize. Consequently, it is too early for this document to become a reference. Instead, it is intended to sit somewhere between a reference and a tutorial that informs and engages a modeler and enables them to create their own simulations. In a longer timeframe, the `pram` package will become one of the final stages in an automated and semi-automated model curation workflow. This document describes the `pram` package v0.1.0.

For illustrative purposes, this document contains a decent amount of code. Most of the code snippets are runnable on their own. Consequently, code can be copied-and-pasted, albeit with a caveat. Care should be taken because whitespace characters (i.e., spaces and end-line characters) are not always copied correctly. This is of particular importance for indentation-based languages such as Python. While substantial effort has been put into containing this phenomenon, the problem afflicts different platforms and different PDF readers to a varying and unpredictable degree and seems non-addressable in a general sense.

The remainder of this document is structured as follows. In Section 1, we briefly motivate the choice of the programming language and, to make the rest of this report more accessible, glide over the basic concepts related to the implementation. In Section 2, we address the modularity of this software by describing its major components. In Section 3 and Section 4, we discuss obtaining and running the code. Finally, in Section 5, we dissect several sample simulations.

[*]School of Computing and Information, University of Pittsburgh; tomek.loboda@gmail.com

Contents

1 Introduction

1.1 Principles

This software is being developed with the following principles in mind:

- **Simplicity.** The code should be as simple and as succinct as possible.
- **Readability.** The code should be highly readable (beyond but including self-documenting).
- **Maintainability.** The code should be easily modified and extended.
- **Elegance.** The code should be well-designed.
- **Generality.** The code should cater to a vast range of simulation scenarios.
- **Late optimization.** "Premature optimization is the root of all evil." [2]

Combining the above principles ensures high quality and longevity of the codebase.

1.2 Programming Paradigms

This software is being implemented in the Python programming language. Apart from being the *de facto* standard for scientific and research computing, Python comes with "batteries included." Moreover, Python is a multi-paradigm programming language that encompasses both procedural and object-oriented programming (OOP) paradigms, among others. In this software, we embrace the object-oriented approach because it promotes two important aspects of codebase design:

- *Code composition* (i.e., the solution implemented is broken down into distinct but interacting classes which may or may not be structurally related to one another), and
- *Encapsulation* (which binds data and functions that manipulate that data together in objects).

More can be said about the benefits of OOP, but we will err on the side of brevity. An astute reader will notice that a highly dynamic language like Python does not actually promise proper encapsulation in the sense of the entrails of an object being inaccessible from its outside, but the fact remains that all bits of code relevant to an object are kept inside of that object.

Two concepts central to OOP, and relevant to this report, are the ones of *a class* and *an object*. A class can be thought of as a blueprint for an object in that while the software is running, multiple objects (or *instances*) of the same class can be present. We will see this concept applied later on when multiple groups of agents will be created from a single class definition.

Before we move on to describing the main classes of `pram`, one additional mention needs to be made. While the previous reports referred to group characteristics as "features and relations", the present implementations uses the terms "attributes and relations" which, when written in code, is shortened to `attr` an `rel`.

2 Main Classes

2.1 Class Diagram

Figure 1 shows a high-level view on how `pram` has been designed. In OOP, *a base class* (or *superclass*) is *extended* by one or more *subclasses*. Another way of saying this is that a subclass *inherits* from a base class. An example of inheritance in Figure 1 is `Agent` being an `Entity`.

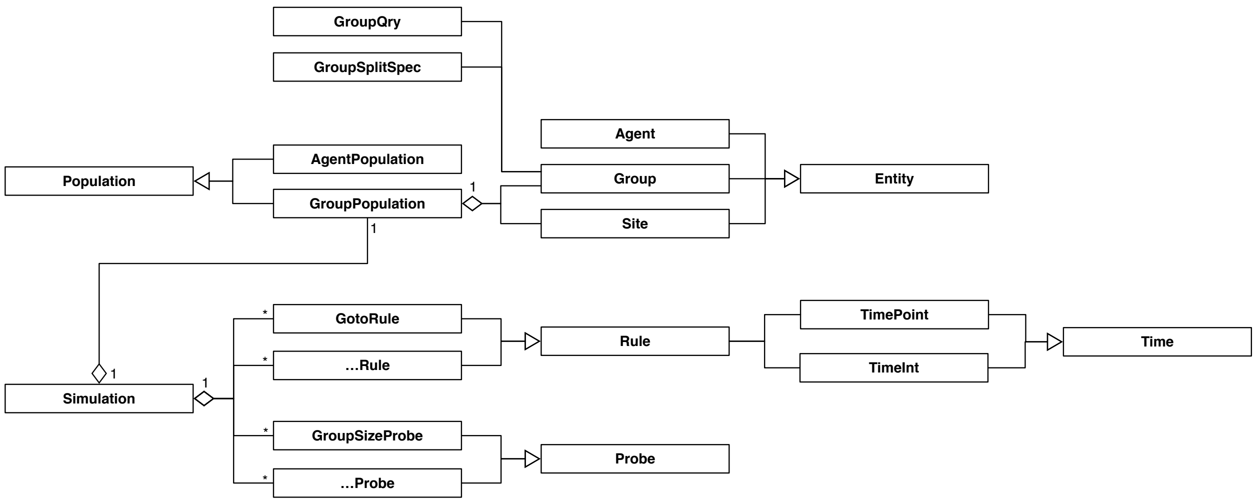

Figure 1: Class diagram of the `pram` package.

In a Chess program, the `Pawn` class would extend the `Piece` class. This is sometimes referred to as *specialization* because the subclass is less general than the base class. For example, any `Piece` has a `move()` function, but only `Pawn` has the `promote()` function. Functions attached to classes and objects are called *methods*.

2.2 Simulation

This is the main and most important user-facing class of `pram`. `pram` conceptualizes a simulation as a sequence of steps where at each step rules are applied to groups. As we will see later, not every rule may fire at every simulation step and even if a rule fires it may not apply to every group. In fact, it is one of the tasks of the modeler to design the group-rule dynamics of a simulation.

The meaning of a single simulation step is arbitrary and the modeler has the freedom to mold the notion of time to the requirements of their particular problem. That is, while the time is discrete[1], it is also unit-less; it is the simulation context that defines the appropriate granularity. For instance, million years (myr) might be appropriate for geological processes while Plank time might be appropriate for modeling quantum phenomena.

For a quick example, the following code creates a simulation that starts at 6am, has a step size of one hour, and runs for 24 consecutive hours:

```
from pram.sim import Simulation
s = Simulation(6,1,24, rand_seed=1001)
```

However, this code could also imply a simulation that starts on the 6th of March, has a step size of one day and runs for 24 days. Again, defining time semantics is part of modeling.

The `rand_seed` argument in the code snippet above is optional; results of different simulations with the same random seed are guaranteed to be identical (provided they don't depend on external factors that may change between runs). That makes this argument useful for testing. In more technical terms, random seed determines the sequence of pseudo-random numbers which are often used to imbue simulations with randomness inherent to the real world. For example, a simulation might need to generate a population of agents with certain heights and weights; those heights and weights would be pseudo-random draws from a pair of distributions, one for heights (e.g., $\mathcal{N}(160, 20)$ cm) and one for weights (e.g., $\mathcal{N}(80, 15)$ kg), although in more realistic models parameters of these distributions would depend on other variables (e.g., age and gender).

[1]We may entertain the idea of continuous time at some point in the future, possibly via systems of differential equations.

2.3 `GroupPopulation`

This class would ordinarily not be used by the modeler directly. It is however used under the hood and serves an important purpose of a container for entities that comprise a population (e.g. groups) or are relevant to a population (e.g., sites) as well as functionality to conveniently and safely manipulate them (e.g., query groups). It is this class that in the future iterations of this software will convert between agent-based and group-based simulations thus providing integration with other simulation frameworks, such as FRED [1].

The following code snippet adds a group of a thousand agents to a `Simulation` object, but internally it is the `GroupPopulation` object that handles this operation:

```
from pram.sim import Simulation
s = Simulation(6,1,24)
s.create_group(1000)
```

As we will see in Section 5, another, more explicit and therefore preferred way of adding groups exists.

2.4 `AgentPopulation`

At this point, this is a placeholder class that will handle populations of agents once interoperability with FRED (and, possibly, other agent-based simulation frameworks) has been established.

2.5 `Entity`

An abstract class (i.e., one that cannot be instantiated) that is the base class for all entities. The entities implemented at this point are `Agent` (Section 2.6), `Site` (Section 2.7), and `Group` (Section 2.8).[2] Entities are *nouns* of PRAM simulations.

[2]In future iterations, a `Resource` entity might make an appearance. A resource would be an entity shared by agents and as such would accommodate a finite number of agents. A good example of a resource is a public bus, the cornerstone of the public transit system. This sort of an entity, having a clear mapping to the real world, would certainly enhance the expressiveness of simulations and perhaps impact their dynamics.

2.6 Agent

An entity representing a singular agent, as used in agent-based modeling. Depending on the simulation context, an agent can be a synthetic proxy for a particle in a fluid, a molecule, or a human being. Because the current version of the software focuses exclusively on groups (i.e., we are still ensuring our representation is expressive enough and our algorithm works as intended), this class does not directly participate in simulations and will remain dormant until integration with other agent-based modeling frameworks.

As a placeholder class, `Agent` has several hard-coded attributes, which is useful for testing purposes. A more general implementation will be proposed in the future. The following code snippet instantiates a sample `Agent` object and prints its information.

```python
from pram.entity import Agent, AttrSex
print(Agent('smith', AttrSex.M, 99, school=None, work='matrix', location=''))
```

```
Agent  name: smith          sex:M  age:  99  flu: NO      ...
```

Agents with random characteristics (i.e., random draws from user-defined probability distributions) can also be generated, both one-by-one and on a batch-basis:

```python
import numpy as np
from pram.entity import Agent

np.random.seed(1009)

print(Agent.gen('duncan'))
print(Agent.gen('irulan'))

for a in Agent.gen_lst(5):
    print(a)
```

```
Agent  name: duncan         sex:M  age:  33  flu: SYMPT   ...
Agent  name: irulan         sex:F  age:  27  flu: SYMPT   ...
Agent  name: a.0            sex:M  age:  32  flu: ASYMPT  ...
Agent  name: a.1            sex:F  age:  34  flu: ASYMPT  ...
Agent  name: a.2            sex:M  age:  51  flu: SYMPT   ...
Agent  name: a.3            sex:F  age:  62  flu: SYMPT   ...
Agent  name: a.4            sex:M  age:  48  flu: SYMPT   ...
```

2.7 `Site`

An entity representing a physical location, such as a school or a store, where agents may reside and co-reside with other agents. The following code snippet creates a dictionary (which is equivalent to a hash table or hash array in other languages) of two sites and prints information about one of them:

```
from pram.entity import Site
sites = {
    'home': Site('h'),
    'work': Site('w')
}
print(sites['home'])
```

```
Site  name: h             hash: 4160006706868298494
```

The `Site` class has two methods that enable the modeler to condition on the population currently located at a site (more methods will be added later):

```
from pram.entity import GroupQry, Site
from pram.pop import GroupPopulation

s = Site('store')
s.set_pop(GroupPopulation())
...
s.get_pop_size()
s.get_groups_here(GroupQry({ 'is-student': True }, { 'location': Site('pitt') }))
```

The modeler can also define their own site by extending the `Site` class. For example:

```
from pram.entity import Site
from pram.pop    import GroupPopulation

class GeoSite(Site):
    def __init__(self, name, geo_coord=(0.0, 0.0), pop=None):
        super().__init__(name, pop=pop)
        self.geo_coord = geo_coord

    def print_location(self):
        print('{} is on Earth.'.format(self.name))  # 'self.name' is a Site's property

s = GeoSite('pittsburgh', (40.440624, -79.995888), GroupPopulation())
s.get_pop_size()    # method from Site
s.print_location()  # method from GeoSite
```

There is no danger that this will break `pram` because the instance of the `GeoSite` class is, by definition, also an instance of the `Site` class. Naturally, if the modeler overloads base class members (properties or methods) that `pram` relies on, that may result in broken code. More compelling examples of inheritance will be presented as the software matures.

The `Site` class contains a useful constant[3] named `AT`. That constant should be used whenever the current location of a group should be retrieved or set. For example, the following code snippet

[3] In actuality, there are no constants in Python, but by a naming convention all-uppercase symbols (e.g., `AGE = 7`) are considered constants and should not be changed.

creates an empty group (i.e., one with size of zero) and a single relation which denotes that the group is currently shopping at Trader Joe's:

```python
from pram.entity import Group, Site
Group(rel={ Site.AT: Site('Trader Joe')})
```

Future iterations of the package may introduce other constants.

2.8 Group

An entity representing a group of agents; agents within a group as assumed indistinguishable from one another. Any two groups are assumed identical if they have identical sets of attributes and relations. Apart from attributes and relations, a group is characterized by a name and size. For example, the following code snippet creates a group of 200 gardeners who attend the University of Pittsburgh.

```
from pram.entity import Group
g = Group('g0', n=200, attr={ 'occup': 'gardener' }, rel={ 'school': 'pitt' })
```

All argument are optional. Below are several more examples of how groups can be instantiated:

```
from pram.entity import Group

Group()
Group('g0')
Group(n=200)
Group(rel={ 'school': 'pitt' })
```

The modeler is unlikely to instantiate groups this way when building a simulation; we will see an alternative way in Section 5.

The `Group` class has several methods that a modeler would make an extensive use of when constructing rules within a simulations.

```
# Checking if a group has one or more attributes irrespective of their values:
g.has_attr('occup')
g.has_attr([ 'occup' ])
g.has_attr([ 'occup', 'children-cnt' ])

# Checking if a group has one or more attributes with particular values:
g.has_attr({ 'occup': 'librarian' })
g.has_attr({ 'occup': 'librarian', 'children-cnt': 3 })

# The same check are available for relations:
g.has_rel({ 'school': 'pitt' })

# Getting values of attributes and relations:
g.get_attr('occup')
g.get_rel('school')
```

The `Group` class has group-altering methods as well: `set_attr()`, `set_attrs()`, `set_rel()`, `set_rels()`. While useful for programmatically creating groups, they will likely never be used directly after a simulation has started. That is, those methods are not useful for rule building, because rules perform agent mass redistribution via group splitting which is scheduled with the `GroupSplitSpec` class that we describe next.

After `Simulation` and `Rule`, `Group` is the third most important class a modeler will use.

2.9 GroupSplitSpec

This class defines how a group is split into other groups and is the basic mediator of population mass distribution. The `GroupSplitSpec` constructor's[4] signature reveals that the modeler controls mass distribution by indicating which attributes and relations should be set and which deleted. We will illustrate this with the following example describing a hypothetical case of 2% of a population of students graduating from a university and moving on to work at a conservation organization:

```
from enum import IntEnum
from pram.entity import GroupSplitSpec, Site

Degree = IntEnum('Degree', 'NONE BA BS MA MS PHD')

GroupSplitSpec(
    p=0.02,
    attr_set={ 'degree': Degree.MS },                       # a dict
    attr_del=set({ 'is-student' }),                         # a set
    rel_set={ 'work': Site('Polar Bears International') },  # a dict
    rel_del=set({ 'school' })                               # a set
)
```

More context will be provided in Section 2.12, but to foreshadow the discussion of the `Rule` class, a modeler will typically control mass distribution via a list of `GroupSplitSpec` objects, ensuring that the probabilities add up to one:

```
from pram.entity import GroupSplitSpec
from pram.rule   import AttrFluStatus

p = 0.05  # prob of infection

[
    GroupSplitSpec(p=1 - p, attr_set={ 'flu-stage': AttrFluStage.NO     }),
    GroupSplitSpec(p=p,     attr_set={ 'flu-stage': AttrFluStage.ASYMPT })
]
```

We will see a larger piece of this code in the Progression of the Flu simulation in Section 5.2.

There is no limit to how many groups may result from splitting a group and `pram` makes sure to create any non-extant groups as needed. In fact, this group-splitting mechanism can be used to move the entire population mass from one group to another group. While the usage of the word "split" may seem unwarranted here in that no actual splitting happens, the underlying mechanism is the same. The uniformity of how mass distribution is handled is an important forte of this approach. Below is an example of how the entire group's population can be moved to another group:

```
[GroupSplitSpec(p=1.0, attr_set={ 'did-attend-school-today': False })]
```

The critical part is that the probability of the population mass from the current group moving to the group with the altered set of attributes is `1.0`. Note that the split specification is still a list. We will revisit this particular example when discussing the `Rule` class in Section 2.12.

[4] A constructor is a method that is called automatically when an object is being instantiated. It is a good practice to perform all object initialization in the constructor. In Python, a constructor is the method named `__init__()` and it is one of the so-called magic methods.

2.10 `GroupQry`

While building a simulation, the modeler might be interested in a subset of groups that meets certain criteria with respect to attribute and relations. The `GroupQry` provides the means of selecting those group subsets. This is how the class is instantiated:

```
from pram.entity import GroupQry
GroupQry(attr={}, rel={})
```

The API of `pram` uses `GroupQry` objects in multiple places. For example, the code snippet below demonstrates how to get sizes of groups of students currently located at the Pittsburgh Symphony Orchestra.

```
from pram.entity import GroupQry, Site

pso = Site('Pittsburgh Symphony Orchestra')
...
groups = pso.get_groups_here(GroupQry({ 'is-student': True }, { 'school': Site('pitt') }))
for g in groups:
    print(g.size)
```

Apart from being an important tool in the modeler's arsenal, the `GroupQry` class is also used internally by the package.

In its current shape, the `GroupQry` class is insufficiently flexible and will be updated in the future. For the time being, the modeler will need to use the `Group` class' interface (i.e., `has_attr()` and `has_rel()`) to construct more elaborate queries.

We end this subsection with a more complete piece of code which shows various ways of querying groups.

```python
from pram.entity import Group, GroupQry, Site
from pram.pop    import GroupPopulation

pop = GroupPopulation()

s = Site('Pittsburgh Symphony Orchestra', pop=pop)

g1 = Group(n=1000, attr={ 'is-student': True                 }, rel={ Site.AT: s })
g2 = Group(n= 200, attr={ 'is-student': True, 'major': 'CS'  }, rel={ Site.AT: s })
g3 = Group(n=  30, attr={ 'is-student': True, 'major': 'EPI' }, rel={ Site.AT: s })
g4 = Group(n=   4, attr={ 'is-student': False                }, rel={ Site.AT: s })

pop.add_groups([g1, g2, g3, g4])

n1 = sum([g.n for g in s.get_groups_here()])
n2 = sum([g.n for g in s.get_groups_here(GroupQry())])
n3 = s.get_pop_size()
print('{} {} {}'.format(n1, n2, n3))  # output: 1234.0 1234.0 1234.0

n4 = sum([g.n for g in s.get_groups_here(GroupQry({ 'is-student': True }))])
print('{}'.format(n4))  # output: 1230.0

n5 = sum([g.n for g in s.get_groups_here() if g.has_attr('major')])
print('{}'.format(n5))  # output: 230.0

n6 = sum([g.n for g in s.get_groups_here(GroupQry({ 'is-student': True, 'major': 'CS' }))])
print('{}'.format(n6))  # output: 200.0

n7 = sum([g.n for g in s.get_groups_here(GroupQry({ 'is-student': True, 'major': 'PHIL' }))])
print('{}'.format(n7))  # output: 0
```

```
1234.0 1234.0 1234.0
1230.0
230.0
200.0
0
```

2.11 `Time`, `TimePoint`, and `TimeInt`

Because the granularity of time is under the modeler's control, it is important for them to be able to also control the exact times when each of the simulation rules applies. The three classes described in this subsection is how the `pram` package enables that.

The `Time` class is the base class which is not used directly and needs to be extended. As for the other two classes, `TimePoint` defines a single point in time while `TimeInt` defines a time interval. Because these classes are intricately related to rules, they reside in the `pram.rule` namespace, but this may change in the future. Below are examples of usage of both of them.

```
from pram.rule import TimeInt, TimePoint
t1 = TimePoint(8)    # 8am, 8th day of the month, etc.
t2 = TimeInt(10,22)  # from 10am to 10pm (24h clock)
```

We will see how these time specifications are used in the following subsections. Future iterations of the package will expand on time specification and handling in good ways.

2.12 Rule

In terms of importance for the modeler, Rule is second only to Simulation. It is also the class that will need the most time and attention.

Being *verbs*, rules animate PRAM simulations and that makes it difficult to describe them without proper context. What will make rules really come to life are the sample simulations: Progression of The Flu (Section 5.2) and Attending School (Section 5.3). Consequently, what we will do in this subsection is look at the structure of rules and how that is relevant to the internal workings of PRAM.

Structurally, rules are fairly simple; there are three methods a modeler needs to be aware of: setup(), is_applicable(), and apply().

setup()

The setup() method is executed before the main simulation loop and is intended to initialize the group population. For example, if a rule depends on groups having a particular set of attributes and relations, those attributes and relations should be added at this stage.

To foreshadow the detailed treatment of the Attending School simulation, below is an example of a setup() method. This particular method uses the group splitting mechanism to move population of every group to a group that is identical to that group but also has the { 'did-attend-school-today': False } attribute. If such a group does not already exist, it will be created. Either way, the current's group population will be moved in its entirety to the other group.

```
@staticmethod
def setup(pop, group):
    return [GroupSplitSpec(p=1.0, attr_set={ 'did-attend-school-today': False })]
```

is_applicable()

The is_applicable() method is called at every simulation step and ensures that the rule will be applied only in the specified context. The current version of the pram package defines context as a pair of (a) the group and (b) the current simulation timer. To make another premature reference, but this time to the Progression of The Flu simulation, the method below will declare a rule applicable if the “time is right” and the group has the flu-stage attribute defined (with the value of that attribute being irrelevant).

```
def is_applicable(self, group, t):
    return super().is_applicable(t) and group.has_attr([ 'flu-stage' ])
```

For example, only the last three of the following six groups would be compatible with this rule and therefore the rule would not make any changes to the first three groups:

```
# not applicable:
Group()
Group(attr={ 'is-student': True })
Group(rel={ 'home': Site('h') })

# applicable:
Group(attr={ 'flu-stage': None })
Group(attr={ 'flu-stage': 'asymptomatic' })
Group(attr={ 'flu-stage': 'symptomatic', 'is-student': True }, rel={ 'home': Site('h') })
```

The term "time is right" goes back to time specification using the `TimePoint` and `TimeInt` classes we have discussed in Section 2.11. Intuitively, if the current simulation timer falls within a time specification, then the rule is applicable based on the time criterion.

`pram` automatically checks if a rule is applicable to the current context before it applies, so there is no need for a modeler to worry about that.

`apply()`

The `apply()` method is where mass distribution happens. For example, borrowing literarily from Section 5.2, the following code will distribute the population mass contingent upon the current `flu-stage` of the group and three transition probability distributions, one of which depends on the unconditional probability of infection.

```
def apply(self, pop, group, t):
    p = 0.05  # prob of infection

    if group.get_attr('flu-stage') == AttrFluStage.NO:
        return [
            GroupSplitSpec(p=1 - p, attr_set={ 'flu-stage': AttrFluStage.NO     }),
            GroupSplitSpec(p=p,     attr_set={ 'flu-stage': AttrFluStage.ASYMPT }),
            GroupSplitSpec(p=0.00,  attr_set={ 'flu-stage': AttrFluStage.SYMPT  })
        ]
    elif group.get_attr('flu-stage') == AttrFluStage.ASYMPT:
        return [
            GroupSplitSpec(p=0.20, attr_set={ 'flu-stage': AttrFluStage.NO     }),
            GroupSplitSpec(p=0.00, attr_set={ 'flu-stage': AttrFluStage.ASYMPT }),
            GroupSplitSpec(p=0.80, attr_set={ 'flu-stage': AttrFluStage.SYMPT  })
        ]
    elif group.get_attr('flu-stage') == AttrFluStage.SYMPT:
        return [
            GroupSplitSpec(p=0.05, attr_set={ 'flu-stage': AttrFluStage.NO     }),
            GroupSplitSpec(p=0.20, attr_set={ 'flu-stage': AttrFluStage.ASYMPT }),
            GroupSplitSpec(p=0.75, attr_set={ 'flu-stage': AttrFluStage.SYMPT  })
        ]
```

2.13 GotoRule

This is an example of a rule derived from the `Rule` class. We include it in the package because it is general enough to be useful in a variety of situations. Future releases will likely contain other elementary building block rules like that.

Objects of the `GotoRule` class change the current location of a specified portion of the population. For example, the following rule compels 40% of agents that are currently at home to go to work.

```python
from pram.rule GotoRule, TimeInt
GotoRule(TimeInt(8,12), 0.4, 'home', 'work', 'Some agents leave home to go to work')
```

By inspecting the source code of the package (`src/pram/rule.py`) we find that, intuitively, for this rule to be applicable to a group, apart from the usual time criterion, a group has to have relations `home` and `work` defined, and its current location (i.e., relation `Site.AT`) must be the site defined as `home`. In that respect, this rule does not care where the particular group lives or where it works, as long as members of that group have a home and a workplace.

The `GotoRule` can also send populations to a destination site irrespective of their current location. For example, the following instance of the `GotoRule` class sets the location of all groups to the site defined as their `home`:

```python
from pram.rule GotoRule, TimePoint
GotoRule(TimePoint(2), 1.0, None, 'home', 'All still-working agents return home')
```

2.14 `Probe` and `GroupSizeProbe`

The `Probe` class is an abstract class that provides an interface for recording values of variables of interest. This is the base class of what will become a flexible and powerful live simulations data collection solution.

At this point, the only class that extends the base class is `GroupSizeProbe`. This probe collects information about sizes of groups the list of which is provided by the modeler. The following example uses the object of this class to sample population mass distribution sliced by the `flu-stage` attribute:

```
from pram.data   import GroupSizeProbe
from pram.entity import GroupQry
from pram.rule   import AttrFluStage

GroupSizeProbe('flu', [GroupQry(attr={ 'flu-stage': fs }) for fs in AttrFluStage])
```

Notice how the `GroupQry` class described in Section 2.10 is used to specify conditions a group needs to meet in order to be selected by the probe. More elaborate queries can be created this way. This mechanism will become even more important for interrogating the internal state of simulations as the package continues to evolve.

A similar piece of code can be used to sample population mass distribution sliced by its current location (i.e., the `Site.AT` relation):

```
from pram.data   import GroupSizeProbe
from pram.entity import GroupQry, Site

sites = {
    'home': Site('h'),
    'work': Site('w')
}

GroupSizeProbe('site', [GroupQry(attr={ Site.AT: s }) for s in sites])
```

The two "syntactic sugar" methods, `by_attr()` and `by_rel()`, help to get the same result more succinctly, without setting up an explicit iteration (both these methods make an appearance in sample simulations in Section 5):

```
from pram.data import GroupSizeProbe
from pram.rule import AttrFluStatus

GroupSizeProbe.by_attr('flu', 'flu-status', AttrFluStatus)
```

```
from pram.data   import GroupSizeProbe
from pram.entity import Site

sites = {
    'home': Site('h'),
    'work': Site('w')
}

GroupSizeProbe.by_rel('site', Site.AT, sites.values(), memo='Mass distribution across sites')
```

The `memo` argument above is used to provide a clear description of the probe's purpose. Note that the two above "shortcut" methods will only be helpful in cases where only one attribute (or relation) are of interest and an *iterable*[5] of the values of that attribute (or relation) can be made available.

[5]An object capable of returning its members one at a time, irrespective of its actual data type. Tuples, lists, and sets are example of data types that implement the iterable protocol. More information at: `https://docs.python.org/3.6/glossary.html#term-iterable`.

3 Getting the Code

The source code of the `pram` package (and the sample simulations code that uses the package) resides on GitHub under the following URL:

https://github.com/momacs/pram

The entirety of the codebase can be viewed in an internet browser so for cursory inspection no download is necessary. However, in this section, we show how to download the code to your local machine.

The easiest way to get the source code is to clone the GitHub repository. This can be done via an internet browser or via the command line.

3.1 Via Internet Browser

To clone the repository via a browser:

1. Navigate to https://github.com/momacs/pram.
2. Click the green "Clone or Download" button located on the right side of the page.
3. Select "Download ZIP" from the small popup pane that appears.

3.2 Via Command Line

To clone the repository via the command line, run the following command:

```
git clone https://github.com/momacs/pram
```

This will create a directory named `pram` in your current working directory and place the contents of the repository inside it.

Some operating systems (e.g., MacOS) come with the `git` program installed. It may however need to be installed on others:

```
sudo pkg install git   # FreeBSD
sudo apt install git   # Ubuntu
```

`git` for Windows can be downloaded from https://git-scm.com/downloads.

4 Getting and Running the Code

One of the inconveniences of having multiple software projects reside on the same machine may come in the form of libraries coming into conflict with one another. For example project X and project Y may require different and incompatible versions of the same library Z. It is often unclear how to resolve such conflicts, and a resolution may even be impossible. Python's virtual environments (or `venvs`) address this problem by offering a mechanism to encapsulate a project inside of its own runtime environment. That environment is isolated from any other environment and, because `venvs` don't interact, dependencies of every project can be easily and safely satisfied. In this section, we will set up such an environment and download the `pram` package v0.1.0 into it.

Virtual environments can be created by the way of the Python or Anaconda. The Python way is standard and pure, independent or third-party solutions; it is also the way presented here. Anaconda is a software distribution that contains Python, about 3GB of science and data-science related packages, as well as a separate package manager named `conda` (Python's native one is named `pip`; a recursive acronym for "Pip Installs Package"). Anaconda does thing its own quirky way, especially when it comes to virtual environment management, which makes it hard to know what exactly is happening. For that reason, we postpone official support for Anaconda.

The easiest way to setup a virtual environment for the `pram` package is via the `momacs` command-line utility. Instructions on how to obtain the utility can be found on the following page:

https://github.com/momacs/misc

`momacs` supports modern versions of FreeBSD, MacOS, and Ubuntu operating systems.

FreeBSD Consideration
Python virtual environments are not currently supported on FreeBSD due to issues with dependencies (specifically, `scipy` and `numpy`); this issue is not specific to the `pram` package. Consequently, FreeBSD systems need to have `pram` installed in the global interpreter's namespace. Effort will be made to ameliorate this situation.

4.1 Automated Method (FreeBSD, MacOS, and Ubuntu)

Once `momacs` has been installed, a `venv` for `pram` can be created and set up like so:

```
momacs pram setup
```

To activate the `venv` and run a simple simulation, run:

```
source pram/bin/activate
python pram/src/sim_01_simple.py
```

This should produce the following output:

```
 6  flu: (1.00 0.00 0.00 )   ( 1000.0       0       0 )   [1000.0]
 7  flu: (1.00 0.00 0.00 )   ( 1000.0       0       0 )   [1000.0]
 8  flu: (0.95 0.05 0.00 )   (  950.0    50.0       0 )   [1000.0]
 9  flu: (0.91 0.05 0.04 )   (  912.5    47.5    40.0 )   [1000.0]
10  flu: (0.88 0.05 0.07 )   (  878.4    53.6    68.0 )   [1000.0]
11  flu: (0.85 0.06 0.09 )   (  848.6    57.5    93.9 )   [1000.0]
12  flu: (0.82 0.06 0.12 )   (  822.4    61.2   116.4 )   [1000.0]
13  flu: (0.80 0.06 0.14 )   (  799.3    64.4   136.3 )   [1000.0]
14  flu: (0.78 0.07 0.15 )   (  779.0    67.2   153.7 )   [1000.0]
15  flu: (0.76 0.07 0.17 )   (  761.2    69.7   169.1 )   [1000.0]
16  flu: (0.75 0.07 0.18 )   (  745.5    71.9   182.6 )   [1000.0]
17  flu: (0.73 0.07 0.19 )   (  731.8    73.8   194.4 )   [1000.0]
18  flu: (0.72 0.08 0.20 )   (  719.7    75.5   204.9 )   [1000.0]
19  flu: (0.71 0.08 0.21 )   (  709.0    77.0   214.0 )   [1000.0]
20  flu: (0.70 0.08 0.22 )   (  699.7    78.3   222.1 )   [1000.0]
21  flu: (0.70 0.08 0.22 )   (  699.7    78.3   222.1 )   [1000.0]
```

To update a previously set up `venv` with the latest version of `pram`'s source code, execute the following command from the `venv`'s directory (the command will only run in the right directory so there is no danger of filesystem contamination):

```
momacs pram update
```

As an aside, `momacs` can also clone the GitHub repository of `pram`:

```
momacs pram clone
```

If Python 3.6 or higher is not detected, `momacs` will install it, but it can also be downloaded manually from `https://python.org`.

4.2 Manual Method (MacOS and Ubuntu)

The steps to manually set up a virtual environment for `pram` are:

1. Install Python 3.6 or higher.

   Python can either be download from `https://python.org` or installed via the package manager. On Ubuntu the command is:

   ```
   sudo apt install -y python3 python3-pip python3-venv
   ```

   On MacOS, install the HomeBrew package manager and invoke it like so:

   ```
   brew install python3
   ```

2. Install `git`.

   `git` can either be download from `https://git-scm.com` or installed via the package manager. On Ubuntu, the command is:

   ```
   sudo apt install -y git
   ```

   MacOS comes with `git` installed.

3. Create a `venv`.

   ```
   python3 -m venv pram
   ```

4. Pull the GitHub repository into the `venv`.

   ```
   cd pram

   git init
   git remote add origin https://github.com/momacs/pram
   git pull origin master
   ```

5. Activate the `venv` and install dependencies.

   ```
   source bin/activate
   python -m pip install -r requirements.txt
   ```

6. Run a sample simulation.

   ```
   python src/sim_01_simple.py
   ```

4.3 Manual Method (FreeBSD)

As mentioned earlier, attempts to install `pram` in a virtual environment will fail due to `scipy` and `numpy` compilation issues. Consequently, `pram` needs to be run using the global Python interpreter. The installation steps are as follows:

```
sudo pkg install -y git python36 py36-pip py36-scipy py36-numpy

sudo python3.6 -m pip install --upgrade pip
sudo python3.6 -m pip install attrs

git clone https://github.com/momacs/pram
```

A sample simulation can then be run like so:

```
python3.6 pram/src/sim_01_simple.py
```

4.4 Manual Method (Windows)

Official Windows support will be coming in future releases.

4.5 The Future Method

When the `pram` package matures to a stable release (i.e., v1.0.0 and beyond), it will be published to PyPI. That will facilitate the installation by involving the Python's package manager:

```
python -m pip install pram
```

However, our suggestion to use virtual environments will stand.

5 Sample Simulations

This section contains few simple but complete simulations implemented using `pram`. More realistic scenarios hardening the idea of PRAMs as the target language for model curation efforts will arrive later.

5.1 Working and Shopping

This section demonstrates setting up a simulation, defining population groups, defining probes, adding rules, and printing simulation summary. To keep things simple, this simulation uses only one rule class, `GotoRule` (Section 2.13).

5.1.1 Running

```
python pram/src/sim_02_home_work_store.py
```

5.1.2 The Code

Preliminaries

We begin by importing all necessary parts of `pram`:

```
from pram.sim    import Simulation
from pram.entity import GroupQry, Site
from pram.data   import GroupSizeProbe
from pram.rule   import GotoRule, TimeInt, TimePoint
```

Next, we set the random seed (for reproducibility) and define several sites where the population lives, works, and shops. Note that there is only one "home" site which implies that all agents live at the same place. While it may seem strange we would structure our simulation this way, this is just to help keep things manageable by avoiding creating too many `Site` and `Group` objects. Echoing Box's sentiment that "all models are wrong but some are useful," this model is wrong in at least this one way, but it is useful because it allows us to animate the agents and make them go to work and go shopping by the means of distributing population mass among the groups (and creating those groups when necessary).

```
rand_seed = 1928

sites = {
    'home'    : Site('home'),
    'work-a'  : Site('work-a'),
    'work-b'  : Site('work-b'),
    'work-c'  : Site('work-c'),
    'store-a' : Site('store-a'),
    'store-b' : Site('store-b')
}
```

The Probe

Next, we create a probe which, at every step of the simulation, measures and prints the size of the population at each of the six sites:

```
probe_grp_size_site = GroupSizeProbe.by_rel(
    'site', Site.AT, sites.values(), memo='Mass distribution across sites'
)
```

The Simulation

Finally, we set up the simulation. Unlike what we have seen in Section 2, instead of breaking the setup into instantiating individual objects, we use a chaining method call style available in `pram`. Note that in spite of what is true in Python in general, the indentation in this calling style is arbitrary; the one used below lends itself to making the code easier to read.

What we have below is a simulation that starts at 6am and "ticks" every hour for 24 consecutive hours. Furthermore, we have three groups yielding the total population of 2,100 agents and a set of six `GotoRule` rules.

Each of the rules fires either at a specific time (`TimePoint`) or inside of a specific time interval (`TimeInt`) and every rule's purpose is to compel a proportion of agents to change their location. For example, the first rule makes 40% of agents who are at home go to work and fires between 8am and noon. The next-to-last rule fires at midnight and ensures that all shopping agents return home (which coincides with the store closing time assumed by this simulation). The last rule fires at 2am and makes all agents return home to get some sleep, irrespective of where they are (as indicated by the `from` site being `None`).

```
(Simulation(6,1,24, rand_seed=rand_seed).
    new_group('g0', 1000).
        set_rel(Site.AT, sites['home']).
        set_rel('home',  sites['home']).
        set_rel('work',  sites['work-a']).
        set_rel('store', sites['store-a']).
        commit().
    new_group('g1', 1000).
        set_rel(Site.AT, sites['home']).
        set_rel('home',  sites['home']).
        set_rel('work',  sites['work-b']).
        set_rel('store', sites['store-b']).
        commit().
    new_group('g2', 100).
        set_rel(Site.AT, sites['home']).
        set_rel('home',  sites['home']).
        set_rel('work',  sites['work-c']).
        commit().
    add_rule(GotoRule(TimeInt( 8,12), 0.4, 'home',  'work' )).
    add_rule(GotoRule(TimeInt(16,20), 0.4, 'work',  'home' )).
    add_rule(GotoRule(TimeInt(16,21), 0.2, 'home',  'store')).
    add_rule(GotoRule(TimeInt(17,23), 0.3, 'store', 'home' )).
    add_rule(GotoRule(TimePoint(24),  1.0, 'store', 'home' )).
    add_rule(GotoRule(TimePoint( 2),  1.0, None,    'home' )).
    add_probe(probe_grp_size_site).
    summary((True, True, True, True, True), (0,1)).      # output section (1)
    run().                                               # output section (2)
    summary((False, True, False, False, False), (1,1)).  # output section (3)
    run(4)                                               # output section (4)
)
```

5.1.3 The Output

The slightly trimmed version of the output is shown below. We will go through that output in sections, as indicated in the code comments above. Section (1) displays a full summary of the simulation. A summary like this can be requested at any step of a simulation, but in this case is printed before the simulation starts. Requesting a full summary at that point is a practice conducive to making the simulation results easier to reproduce even without the simulation code itself (although a simulation-specific code like custom rules are also needed for that; custom rules are demonstrated in Section 5.2 and Section 5.3).

```
Simulation
    Random seed: 1928
    Timer
        Start      : 6
        Step size  : 1
        Iterations : 24
        Sequence   : [6, 7, 8, 9, 10]
    Population
        Size         : 2100.0
        Groups       : 3
        Groups (ne)  : 3
        Sites        : 6
        Rules        : 6
        Probes       : 1
    Groups (3)
        Group  name: g0  n: 1000.0  attr: {}  rel: {'__at__': Site(home), ... }
        Group  name: g1  n: 1000.0  attr: {}  rel: {'__at__': Site(home), ... }
        Group  name: g2  n:  100.0  attr: {}  rel: {'__at__': Site(home), ... }
    Sites (6)
        Site  name: home     hash: -8704082793172868735
        Site  name: work-a   hash: 6598911341482665850
        Site  name: store-a  hash: -3654620262185175475
        Site  name: work-b   hash: -5748730217739660932
        Site  name: store-b  hash: 4084159153536804648
        Site  name: work-c   hash: 5710016540917094858
    Rules (6)
        Rule  name: goto  t: TimeInt(t0= 8.0, t1=12.0)  p: 0.4  rel: home --> work
        Rule  name: goto  t: TimeInt(t0=16.0, t1=20.0)  p: 0.4  rel: work --> home
        Rule  name: goto  t: TimeInt(t0=16.0, t1=21.0)  p: 0.2  rel: home --> store
        Rule  name: goto  t: TimeInt(t0=17.0, t1=23.0)  p: 0.3  rel: store --> home
        Rule  name: goto  t: TimePoint(t=24.0)  p: 1.0  rel: store --> home
        Rule  name: goto  t: TimePoint(t= 2.0)  p: 1.0  rel: None --> home
    Probes (1)
        Probe  name: site  query-cnt: 6
```

Moving on to section (2) of the output, we see the probe object printing the distribution of mass across the sites (both as a proportion and in absolute numbers). The population mass moves between the sites at proportions and times corresponding to those indicated by the rules.

```
 6  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0      0      0      0      0      0)
 7  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0      0      0      0      0      0)
 8  site: (0.60 0.19 0.19 0.02 0.00 0.00)  ( 1260.0  400.0  400.0   40.0      0      0)
 9  site: (0.36 0.30 0.30 0.03 0.00 0.00)  (  756.0  640.0  640.0   64.0      0      0)
10  site: (0.22 0.37 0.37 0.04 0.00 0.00)  (  453.6  784.0  784.0   78.4      0      0)
11  site: (0.13 0.41 0.41 0.04 0.00 0.00)  (  272.2  870.4  870.4   87.0      0      0)
12  site: (0.08 0.44 0.44 0.04 0.00 0.00)  (  163.3  922.2  922.2   92.2      0      0)
13  site: (0.08 0.44 0.44 0.04 0.00 0.00)  (  163.3  922.2  922.2   92.2      0      0)
14  site: (0.08 0.44 0.44 0.04 0.00 0.00)  (  163.3  922.2  922.2   92.2      0      0)
15  site: (0.08 0.44 0.44 0.04 0.00 0.00)  (  163.3  922.2  922.2   92.2      0      0)
16  site: (0.43 0.26 0.26 0.03 0.01 0.01)  (  906.9  553.3  553.3   55.3   15.6   15.6)
17  site: (0.58 0.16 0.16 0.02 0.05 0.05)  ( 1208.6  332.0  332.0   33.2   97.1   97.1)
18  site: (0.63 0.09 0.09 0.01 0.09 0.09)  ( 1317.4  199.2  199.2   19.9  182.2  182.2)
19  site: (0.64 0.06 0.06 0.01 0.12 0.12)  ( 1346.5  119.5  119.5   12.0  251.2  251.2)
20  site: (0.64 0.03 0.03 0.00 0.14 0.14)  ( 1346.0   71.7   71.7    7.2  301.7  301.7)
21  site: (0.61 0.03 0.03 0.00 0.16 0.16)  ( 1276.4   71.7   71.7    7.2  336.5  336.5)
22  site: (0.70 0.03 0.03 0.00 0.11 0.11)  ( 1478.3   71.7   71.7    7.2  235.6  235.6)
23  site: (0.77 0.03 0.03 0.00 0.08 0.08)  ( 1619.6   71.7   71.7    7.2  164.9  164.9)
 0  site: (0.77 0.03 0.03 0.00 0.08 0.08)  ( 1619.6   71.7   71.7    7.2  164.9  164.9)
 1  site: (0.77 0.03 0.03 0.00 0.08 0.08)  ( 1619.6   71.7   71.7    7.2  164.9  164.9)
 2  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0    0.0    0.0    0.0    0.0    0.0)
 3  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0    0.0    0.0    0.0    0.0    0.0)
 4  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0    0.0    0.0    0.0    0.0    0.0)
 5  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0    0.0    0.0    0.0    0.0    0.0)
```

Section (3) of the output prints the groups that have become part of the simulation after the 24 time steps. The first three groups are the initial ones; the remaining ones were created the first time they were needed to carry population mass.

Evidently, at the end of the simulation, all agents went back to being part of their original groups. To iterate an earlier point, if the agent population lived at multiple home sites, as would be the case in a more realistic scenario, the agents would end up in those home sites (and therefore groups).

For the sake of space, we trim the output, but the reader can run the simulation themselves (`src/sim_02_home_work_store.py`) to inspect the full set of relations in each group.

```
    Groups (8)
        Group  name: g0    n: 1000.0  attr: {}  rel: {'__at__': Site(home), ... }
        Group  name: g1    n: 1000.0  attr: {}  rel: {'__at__': Site(home), ... }
        Group  name: g2    n:  100.0  attr: {}  rel: {'__at__': Site(home), ... }
        Group  name: g0.0  n:    0.0  attr: {}  rel: {'__at__': Site(work-a), ... }
        Group  name: g1.0  n:    0.0  attr: {}  rel: {'__at__': Site(work-b), ... }
        Group  name: g2.0  n:    0.0  attr: {}  rel: {'__at__': Site(work-c), ... }
        Group  name: g0.0  n:    0.0  attr: {}  rel: {'__at__': Site(store-a), ... }
        Group  name: g1.0  n:    0.0  attr: {}  rel: {'__at__': Site(store-b), ... }
```

Finally, section (4) shows four additional steps of the simulation which were run after the initial 24. This is an example of how a modeler could execute an arbitrary number of steps of a simulation. This is also how an HTTP-based interface (e.g., a Web front-end) could be used to control the pace of a simulation interactively.

```
  6  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0     0.0     0.0     0.0     0.0     0.0)
  7  site: (1.00 0.00 0.00 0.00 0.00 0.00)  ( 2100.0     0.0     0.0     0.0     0.0     0.0)
  8  site: (0.60 0.19 0.19 0.02 0.00 0.00)  ( 1260.0   400.0   400.0    40.0     0.0     0.0)
  9  site: (0.36 0.30 0.30 0.03 0.00 0.00)  (  756.0   640.0   640.0    64.0     0.0     0.0)
```

5.2 Progression of The Flu

This section demonstrates aspects the previous simulation touched upon but with an important addition: Defining a custom rule, a task a modeler will perform frequently. What we will aim to establish with this simulation is how the distribution of flu stages changes in a population given a highly simplistic disease progression model.

5.2.1 Running

```
python pram/src/sim_04_flu_progression.py
```

5.2.2 The Code

Preliminaries

We begin by importing all necessary parts of pram and setting the random seed for reproducibility:

```
from pram.sim    import Simulation
from pram.entity import AttrFluStage, GroupSplitSpec
from pram.data   import GroupSizeProbe
from pram.rule   import Rule, TimeInt

rand_seed = 1928
```

The `AttrFluStage` class is an enumeration data type that defines an exhaustive set of mutually exclusive stages an agent susceptible to or infected with the flu can be in: No flu (`NO`), asymptomatic (`ASYMPT`), and symptomatic (`SYMPT`).

The Rule

Our custom rule class extends the `Rule` base class and the constructor doesn't do anything apart from calling the constructor of that base class. By default, the rule fires between the hours of 8am and 8pm, but these values can be changed when the class is instantiated.

In general, if it made sense for a rule object to keep track of something via its internal state, the modeler could elect to initialize that internal state in the rule's constructor.

```
class ProgressFluRule(Rule):
    def __init__(self, t=TimeInt(8,20), memo=None):
        super().__init__('progress-flu', t, memo)
```

The `apply()` method distributes the population mass based on the current flu stage of the group and three transition probability distributions, one of which depends on the unconditional probability of infection (0.05). That is, this model assumes the flu appears spontaneously in a population, plausibly due to factors extraneous to the simulation.

```
    def apply(self, pop, group, t):
        p = 0.05  # prob of infection

        if group.get_attr('flu-stage') == AttrFluStage.NO:
            return [
                GroupSplitSpec(p=1 - p, attr_set={ 'flu-stage': AttrFluStage.NO     }),
                GroupSplitSpec(p=p,     attr_set={ 'flu-stage': AttrFluStage.ASYMPT }),
                GroupSplitSpec(p=0.00,  attr_set={ 'flu-stage': AttrFluStage.SYMPT  })
            ]
        elif group.get_attr('flu-stage') == AttrFluStage.ASYMPT:
            return [
                GroupSplitSpec(p=0.20, attr_set={ 'flu-stage': AttrFluStage.NO     }),
                GroupSplitSpec(p=0.00, attr_set={ 'flu-stage': AttrFluStage.ASYMPT }),
                GroupSplitSpec(p=0.80, attr_set={ 'flu-stage': AttrFluStage.SYMPT  })
            ]
        elif group.get_attr('flu-stage') == AttrFluStage.SYMPT:
            return [
                GroupSplitSpec(p=0.05, attr_set={ 'flu-stage': AttrFluStage.NO     }),
                GroupSplitSpec(p=0.20, attr_set={ 'flu-stage': AttrFluStage.ASYMPT }),
                GroupSplitSpec(p=0.75, attr_set={ 'flu-stage': AttrFluStage.SYMPT  })
            ]
        else:
            raise ValueError("Invalid value for attribute 'flu-stage'.")
```

The is_applicable() method checks if rule should fire based on the simulation timer and whether the group has the flu-stage attribute.

```
    def is_applicable(self, group, t):
        return super().is_applicable(t) and group.has_attr([ 'flu-stage' ])
```

The Probe

Next, we create a probe which, at every step of the simulation, measures and prints the size of population currently in any of the three flu stages:

```
probe_grp_size_flu = GroupSizeProbe.by_attr(
    'flu', 'flu-stage', AttrFluStage, memo='Mass distribution across flu stages'
)
```

The Simulation

Finally, we set up a simulation that starts on the 6th of June, updates once a day, and runs for a total of 16 days. The simulation contains a single group of a 1000 uninfected agents, a single rule, and a single probe.

```
s = Simulation(6,1,16, rand_seed=rand_seed)
s.create_group(1000, { 'flu-stage': AttrFluStage.NO })
s.add_rule(ProgressFluRule())
s.add_probe(probe_grp_size_flu)
s.run()
```

5.2.3 The Output

The output reveals the incidence of the three flu stages in the population over time. This simulation is a time-homogenous Markov chain with a finite state space and the transition probability matrix given later that reaches its stationary distribution $\pi = (0.7 \;\; 0.08 \;\; 0.22)$ on the 20th of June.

```
 6  flu: (1.00 0.00 0.00)   ( 1000.0       0       0)   [1000.0]
 7  flu: (1.00 0.00 0.00)   ( 1000.0       0       0)   [1000.0]
 8  flu: (0.95 0.05 0.00)   (  950.0    50.0       0)   [1000.0]
 9  flu: (0.91 0.05 0.04)   (  912.5    47.5    40.0)   [1000.0]
10  flu: (0.88 0.05 0.07)   (  878.4    53.6    68.0)   [1000.0]
11  flu: (0.85 0.06 0.09)   (  848.6    57.5    93.9)   [1000.0]
12  flu: (0.82 0.06 0.12)   (  822.4    61.2   116.4)   [1000.0]
13  flu: (0.80 0.06 0.14)   (  799.3    64.4   136.3)   [1000.0]
14  flu: (0.78 0.07 0.15)   (  779.0    67.2   153.7)   [1000.0]
15  flu: (0.76 0.07 0.17)   (  761.2    69.7   169.1)   [1000.0]
16  flu: (0.75 0.07 0.18)   (  745.5    71.9   182.6)   [1000.0]
17  flu: (0.73 0.07 0.19)   (  731.8    73.8   194.4)   [1000.0]
18  flu: (0.72 0.08 0.20)   (  719.7    75.5   204.9)   [1000.0]
19  flu: (0.71 0.08 0.21)   (  709.0    77.0   214.0)   [1000.0]
20  flu: (0.70 0.08 0.22)   (  699.7    78.3   222.1)   [1000.0]
21  flu: (0.70 0.08 0.22)   (  699.7    78.3   222.1)   [1000.0]
```

5.2.4 Model Equivalence

The model described above is equivalent to a Dynamic Bayesian network with a single variable `flu` (short for `flu-stage`) that influences itself over time with the initial state distribution

$$\texttt{flu}^{(0)} \quad \begin{array}{c|c} N & 1 \\ A & 0 \\ S & 0 \end{array}$$

and the transition model

$$\begin{array}{cc|ccc} & & & \texttt{flu}^{(t)} & \\ & & N & A & S \\ \hline & N & 0.95 & 0.20 & 0.05 \\ \texttt{flu}^{(t+1)} & A & 0.05 & 0 & 0.20 \\ & S & 0 & 0.80 & 0.75 \end{array}$$

where N, A, and S correspond to the values of the `AttrFluStage` type.

5.3 Attending School

This section demonstrates aspects the previous simulations touched upon but with two custom rules defined. The toy problem we will model is having students attend school. Students will be leaving home to go to school every hour for a few hours. The likelihood of them returning home will increase with the time they spend at school. At some point, all students will return home regardless of how long they've been at school which will simulate the schools closing.

5.3.1 Running

```
python pram/src/sim_03_attend_school.py
```

5.3.2 The Code

Preliminaries

We begin by importing all necessary parts of `pram`:

```
from pram.sim    import Simulation
from pram.entity import GroupQry, GroupSplitSpec, Site
from pram.data   import GroupSizeProbe
from pram.rule   import GotoRule, Rule, TimeInt, TimePoint
```

Next, we set the random seed (for reproducibility) and define a single "home" site and two "school" sites. Just like it was the case earlier, having a single home site is just for expositional convenience.

```
rand_seed = 1928
sites = {
    'home'     : Site('home'),
    'school-a' : Site('school-a'),
    'school-b' : Site('school-b')
}
```

Rule 1

Next, we define our first rule. This rule will fire once a day and "reset" the day so that the simulation continues from the same point the following day. This is not to suggest this is how a simulation should be constructed, but rather to demonstrate that rules can be used to achieve this effect. We will break down our narration by method.

Our custom rule class extends the `Rule` base class and the constructor doesn't do anything apart from calling the constructor of that base class.

```
class ResetDayRule(Rule):
    def __init__(self, t, memo=None):
        super().__init__('reset-day', t, memo)
```

The `apply()` method, from where mass distribution is controlled, uses the group splitting mechanism (see Section 2.9) to set the `did-attent-school-today` attribute of all groups to `False`. This ensures that in the morning, when students are expected to attend school, all groups of agents will be marked as not having been at school yet. The second rule, which we define later, will modify this attribute antagonistically.

```
    def apply(self, pop, group, t):
        return [GroupSplitSpec(p=1.0, attr_set={ 'did-attend-school-today': False })]
```

The `is_applicable()` method checks if rule should fire based on the simulation timer.

```
    def is_applicable(self, group, t):
        return super().is_applicable(t)
```

Rule 2

Next, we define our second rule. Just like before, there is nothing unusual going on in the constructor other than that we define the default time interval to be 8am – 4pm.

```
class AttendSchoolRule(Rule):
    def __init__(self, t=TimeInt(8,16), memo=None):
        super().__init__('attend-school', t, memo)
```

The `apply` method calls the `apply_at_home()` method if the group's current location (i.e., the `Site.AT` relation) is equal to its `home` relation. Alternatively, if the group is currently located at school, the `apply_at_school()` method is called.

```
    def apply(self, pop, group, t):
        if not self.is_applicable(group, t): return None

        if group.has_rel({ Site.AT: group.get_rel('home') }) and (
            not group.has_attr('did-attend-school-today') or
            group.has_attr({ 'did-attend-school-today': False })
        ):
            return self.apply_at_home(group, t)

        if group.has_rel({ Site.AT:  group.get_rel('school') }):
            return self.apply_at_school(group, t)
```

The `apply_at_home()` method sends a proportion of students to the school they attend (because different groups attend different schools). What proportion will be displaced depends on the current simulation time. Specifically, every hour half of the students still at home will go to school and at noon all the students will end up at school.

The way this mechanism is implemented is neither particularly elegant nor particularly general. For example, the use of integers as dictionary keys to hold the probabilities with which students go to school will break when the definition of simulation time changes (e.g., from time passing in 1-hour increments to 30-minute increments). However, this is just an example of rules animate agents in a simulation.

Furthermore, we can see that apart from changing the group's location relation, the method also changes two of its attributes: `did-attend-school-today` and `t-at-school`, the second of which is the number of hours students in that group have spent at school so far.

While this method and the next both result in a binary group splits (i.e., if the rule is applicable to a group, it will split that group in at most two other groups), an arbitrary number of groups can be spawned this way.

```
    def apply_at_home(self, group, t):
        p = { 8:0.50, 9:0.50, 10:0.50, 11:0.50, 12:1.00 }.get(t, 0.00)
            # prob of going to school = f(time of day)

        return [
            GroupSplitSpec(
                p=p,
                attr_set={ 'did-attend-school-today': True, 't-at-school': 0 },
                rel_set={ Site.AT: group.get_rel('school') }
            ),
            GroupSplitSpec(p=1 - p)
        ]
```

The `apply_at_school()` method sends students back home with the probability of that happening being contingent upon the time students have spent at school so far. Specifically, as the time at school increases, so does the probability of going home until eventually, at the 8-hour mark, students are sent home with probability 1. Moreover, if the current simulation time (i.e., `t`) coincides with the end of the rule's time interval (i.e., `self.t.t1`), all students are sent home immediately, which simulates schools closing.

Much like was the case before, inelegance permeates the implementation of this method. Nevertheless, it does what we want it to do and we have something to improve on in the future, always a heart-warming thought.

```
    def apply_at_school(self, group, t):
        t_at_school = group.get_attr('t-at-school')
        p = { 0: 0.00, 1:0.05, 2:0.05, 3:0.25, 4:0.50, 5:0.70, 6:0.80, 7:0.90, 8:1.00 }.get(
            t_at_school, 1.00
        ) if t < self.t.t1 else 1.00  # prob of going home = f(time spent at school)

        return [
            GroupSplitSpec(
                p=p,
                attr_set={ 't-at-school': (t_at_school + 1) },
                rel_set={ Site.AT: group.get_rel('home') }
            ),
            GroupSplitSpec(p=1 - p, attr_set={ 't-at-school': (t_at_school + 1) })
        ]
```

The `is_applicable()` method ensures that the rule will only fire at an appropriate time and only for the appropriate group. That is, only groups with the attribute `is-student` equal to `True` that have the relations `home` and `school` can be affected by this rule. This prevents groups of teachers that may currently be located at a school from being sent home as a result of this rule's application. A different rule would rule the life of teaching agents.

```
    def is_applicable(self, group, t):
        return (
            super().is_applicable(t) and
            group.has_attr({ 'is-student': True }) and
            group.has_rel(['home', 'school']))
```

As a reminder from Section 2.12, the `setup()` method will be called by the simulation object only once and before the actual simulation loop starts. Here, it ensures all groups have the `did-attend-school-today` attribute set to `False`. We see the group splitting mechanism at work again.

Because this way of setting attributes and relations may seem strange to modelers, we will abstract it via a dedicated method in the future.

In a more complicated simulation, this method would not simply alter all groups; instead, it would do it conditionally. For example, it would certainly make sense to set the `did-attend-school-today` attribute only for students (i.e., conditioning on the `is-student` attribute), but in the case of this small simulation it would not make a difference because we only deal with groups of students anyway.

```
    @staticmethod
    def setup(pop, group):
        return [GroupSplitSpec(p=1.0, attr_set={ 'did-attend-school-today': False })]
```

The Probe

Next, we create a probe which, at every step of the simulation, measures and prints the size of population at each of the three sites:

```
probe_grp_size_site = GroupSizeProbe.by_rel(
    'site', Site.AT, sites.values(), memo='Mass distribution across sites'
)
```

The Simulation

Finally, we set up the simulation as starting at 7am and running for 10 steps updating every hour. Additionally, we create two groups of students, each 500 agents strong, and add the two rules and the probe.

```
(Simulation(7,1,10, rand_seed=rand_seed).
    new_group('0', 500).
        set_attr('is-student', True).
        set_rel(Site.AT,  sites['home']).
        set_rel('home',   sites['home']).
        set_rel('school', sites['school-a']).
        commit().
    new_group('1', 500).
        set_attr('is-student', True).
        set_rel(Site.AT,  sites['home']).
        set_rel('home',   sites['home']).
        set_rel('school', sites['school-b']).
        commit().
    add_rule(ResetDayRule(TimePoint(7))).
    add_rule(AttendSchoolRule()).
    add_probe(probe_grp_size_site).
    run()
)
```

5.3.3 The Output

The output is similar to the one from the Working and Shopping simulation we have discussed earlier. Here however we only have three sites.

We can see the agents leaving home starting at 8am. Predictably, at 8am exactly half of the population is still at home. At 9am, that half is cut in half again (recall that this is what rule 2 does). At 10am, however, the proportion of students at home is no longer half of that from 9am due to some students returning home after having attended school. The simulation continues with the smallest number of students at home registering at 11am and everyone back from school at 4pm when the schools close.

```
  7  site: (1.00 0.00 0.00)   ( 1000.0       0       0)   [1000.0]
  8  site: (0.50 0.25 0.25)   (  500.0   250.0   250.0)   [1000.0]
  9  site: (0.25 0.38 0.38)   (  250.0   375.0   375.0)   [1000.0]
 10  site: (0.15 0.42 0.42)   (  150.0   425.0   425.0)   [1000.0]
 11  site: (0.12 0.44 0.44)   (  123.8   438.1   438.1)   [1000.0]
 12  site: (0.19 0.40 0.40)   (  192.2   403.9   403.9)   [1000.0]
 13  site: (0.43 0.29 0.29)   (  426.9   286.6   286.6)   [1000.0]
 14  site: (0.66 0.17 0.17)   (  664.2   167.9   167.9)   [1000.0]
 15  site: (0.82 0.09 0.09)   (  823.4    88.3    88.3)   [1000.0]
 16  site: (1.00 0.00 0.00)   ( 1000.0     0.0     0.0)   [1000.0]
```

5.4 Attending School – Mass Distribution Visualized

To shed more light on mass distribution, we will run several first iterations of the Attending School simulation from the previous subsection and print the groups at the end of each step. The only thing we will modify is the simulation code block; the result will remain the same. As we will soon see the output produced will be far richer.

5.4.1 Running

Comment-out the top simulation and uncomment the bottom simulation in the file below.

```
python pram/src/sim_03_attend_school.py
```

5.4.2 The Code

The simulation code we will use this time is given below and is also available in the Python module `src/sim_03_attend_school.py` (the commented-out portion at the bottom of the file).

```
(Simulation(7,1,10, rand_seed=rand_seed).
    new_group('0', 500).
        set_attr('is-student', True).
        set_rel(Site.AT,  sites['home']).
        set_rel('home',   sites['home']).
        set_rel('school', sites['school-a']).
        commit().
    new_group('1', 500).
        set_attr('is-student', True).
        set_rel(Site.AT,  sites['home']).
        set_rel('home',   sites['home']).
        set_rel('school', sites['school-b']).
        commit().
    add_rule(ResetDayRule(TimePoint(7))).
    add_rule(AttendSchoolRule()).
    add_probe(probe_grp_size_site).
           summary((False, True, False, False, False), (0,1)).
    run(1).summary((False, True, False, False, False), (0,1)).
    run(1).summary((False, True, False, False, False), (0,1)).
    run(1).summary((False, True, False, False, False), (0,1)).
    run(1).summary((False, True, False, False, False), (0,1)).
    run(1).summary((False, True, False, False, False), (0,1))
)
```

5.4.3 The Output

The output produced by this simulation is too wide to be properly reproduced in this document. Consequently, we prune the output significantly while retaining the most import parts of it. What we prune are names of groups, the `is-student` attribute (because it's always the same), and all relations other than the current location (i.e., `__at__`; which is what the `Site.AT` constant resolves to by the way). Moreover, we will shorten the `did-attend-school-today` attribute to `dast`. We retain the site distribution probe to make comparisons between this output and the output from the previous subsection easier.

Step 0

Before the simulation begins, the population consists of the initial two groups we have brought to life.

```
    Groups (2)
        n:  500.0  attr: {}  rel: {'__at__': Site(home) }
        n:  500.0  attr: {}  rel: {'__at__': Site(home) }
```

Step 1

After the first iteration step, we see the result of the `AttendSchoolRule.setup()` method which adds the `dast` attribute to all groups, effectively doubling their number (but notice how population distribution remains the same between the two new groups).

The population mass due to `AttendSchoolRule` has not begun yet, because that rule starts to fire only at 8am, per the `TimeInt(8,16)` time specification.

```
 7  site: (1.00 0.00 0.00)   ( 1000.0       0       0)   [1000.0]
    Groups (4)
        n:      0  attr: {}               rel: {'__at__': Site(home) }
        n:      0  attr: {}               rel: {'__at__': Site(home) }
        n:  500.0  attr: {'dast': False}  rel: {'__at__': Site(home) }
        n:  500.0  attr: {'dast': False}  rel: {'__at__': Site(home) }
```

Step 2

After the second iteration step, we see that half of the mass from each of the initial groups gets distributed to two new groups as a result of the `AttendSchoolRule` sending first batch of students to school. Because no extant groups contain the { `'t-at-school': 0` } attribute, two new groups are created to accommodate the 250 agents from either of the prior groups. Also notice how the two new groups' location reflects the fact that those agents are now at their respective schools.

Another change is that, per `AttendSchoolRule`, the `dast` semaphore variable is set to `True`. Without that, nothing would prevent agents who went to school already and returned home from going back to school the same day. It is up to the modeler to determine whether assumptions like this (here implemented via a boolean variable) should be part of the simulation.

```
 8  site: (0.50 0.25 0.25)   (  500.0   250.0   250.0)   [1000.0]
    Groups (6)
        n:    0.0  attr: {}                                 rel: {'__at__': Site(home) }
        n:    0.0  attr: {}                                 rel: {'__at__': Site(home) }
        n:  250.0  attr: {'dast': False}                    rel: {'__at__': Site(home) }
        n:  250.0  attr: {'dast': False}                    rel: {'__at__': Site(home) }
        n:  250.0  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-a) }
        n:  250.0  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-b) }
```

Step 3

After the third iteration step, the population of agents still at home gets halved again, down to 125 agents per either home-staying group, with the other two portions of 125 agents being added to the already extant groups with the { `'t-at-school': 0` } attribute. Moreover, the population of

500 agents that at this point have been at school for one hour gets moved to two new groups with the { `'t-at-school': 1` } attribute to reflect their accumulating tenure.

Note that no agents have returned home yet per the probabilities specified in the `AttendSchoolRule.apply_at_school()` method. That is about to change though.

```
 9  site: (0.25 0.38 0.38)   (  250.0   375.0   375.0)   [1000.0]
    Groups (10)
        n:    0.0  attr: {}                                rel: {'__at__': Site(home) }
        n:    0.0  attr: {}                                rel: {'__at__': Site(home) }
        n:  125.0  attr: {'dast': False}                   rel: {'__at__': Site(home) }
        n:  125.0  attr: {'dast': False}                   rel: {'__at__': Site(home) }
        n:  125.0  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-a) }
        n:  125.0  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-b) }
        n:    0.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(home) }
        n:  250.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(school-a) }
        n:    0.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(home) }
        n:  250.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(school-b) }
```

Step 4

After the fourth iteration step, the same progression continues, with some half of home-bound agents attending school, and school-bound agents accruing another hour of education.

However, this is also the first time that the `AttendSchoolRule.apply_at_school()` method allows for some students to return home. Specifically, students who have been at school for one hour have a 0.05 probability of going home which is reflected in the sizes of two of the groups becoming 12.5. Let us break this calculation down.

250 agents from `school-a` and another 250 agents from `school-b` have been at school for one hour by this time. Therefore, we get $250 \times 0.05 = 12.5$ from either of those groups to contribute a total of 25 agents who return home (albeit to different groups because their schools are different). As is evident from the sizes of two of the other groups (i.e., those with the { `'t-at-school': 2` } attribute), those 25 home-going agents would otherwise be a part of the groups with the current size of 237.5.

```
10  site: (0.15 0.42 0.42)   (  150.0   425.0   425.0)   [1000.0]
    Groups (14)
        n:    0.0  attr: {}                                rel: {'__at__': Site(home) }
        n:    0.0  attr: {}                                rel: {'__at__': Site(home) }
        n:   62.5  attr: {'dast': False}                   rel: {'__at__': Site(home) }
        n:   62.5  attr: {'dast': False}                   rel: {'__at__': Site(home) }
        n:   62.5  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-a) }
        n:   62.5  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-b) }
        n:    0.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(home) }
        n:  125.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(school-a) }
        n:    0.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(home) }
        n:  125.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(school-b) }
        n:   12.5  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(home) }
        n:  237.5  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(school-a) }
        n:   12.5  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(home) }
        n:  237.5  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(school-b) }
```

Step 5

The simulation goes on with the population mass being continually distributed according to the probabilities defined in the rules.

```
11  site: (0.12 0.44 0.44)   (  123.8   438.1   438.1)   [1000.0]
    Groups (18)
        n:    0.0  attr: {}                                rel: {'__at__': Site(home) }
        n:    0.0  attr: {}                                rel: {'__at__': Site(home) }
        n:  31.25  attr: {'dast': False}                   rel: {'__at__': Site(home) }
        n:  31.25  attr: {'dast': False}                   rel: {'__at__': Site(home) }
        n:  31.25  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-a) }
        n:  31.25  attr: {'dast': True, 't-at-school': 0}  rel: {'__at__': Site(school-b) }
        n:    0.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(home) }
        n:   62.5  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(school-a) }
        n:    0.0  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(home) }
        n:   62.5  attr: {'dast': True, 't-at-school': 1}  rel: {'__at__': Site(school-b) }
        n:  18.75  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(home) }
        n: 118.75  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(school-a) }
        n:  18.75  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(home) }
        n: 118.75  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(school-b) }
        n:  11.88  attr: {'dast': True, 't-at-school': 3}  rel: {'__at__': Site(home) }
        n: 225.62  attr: {'dast': True, 't-at-school': 3}  rel: {'__at__': Site(school-a) }
        n:  11.88  attr: {'dast': True, 't-at-school': 3}  rel: {'__at__': Site(home) }
        n: 225.62  attr: {'dast': True, 't-at-school': 3}  rel: {'__at__': Site(school-b) }
```

Step 10

After 10 iterations, the population consists of 38 groups, most of them empty (not listed here). The reason why agents did not return to their initial groups is that they are not the same agents any more. They attended school, hopefully learned something new (just like we did), and while being back at the same "home" site they came from, they are now distributed according to the number of hours of education they have received so far. Perhaps in a grander simulation these hours of education could be conditioned on to investigate the possible impacts education, or lack thereof, has on the various aspects of a person's life and the society as a whole.

```
16  site: (1.00 0.00 0.00)   ( 1000.0     0.0     0.0)   [1000.0]
    Groups (38)
        n:   25.0  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(home) }
        n:   25.0  attr: {'dast': True, 't-at-school': 2}  rel: {'__at__': Site(home) }
        n:  23.75  attr: {'dast': True, 't-at-school': 3}  rel: {'__at__': Site(home) }
        n:  23.75  attr: {'dast': True, 't-at-school': 3}  rel: {'__at__': Site(home) }
        n: 133.96  attr: {'dast': True, 't-at-school': 4}  rel: {'__at__': Site(home) }
        n: 133.96  attr: {'dast': True, 't-at-school': 4}  rel: {'__at__': Site(home) }
        n: 169.22  attr: {'dast': True, 't-at-school': 5}  rel: {'__at__': Site(home) }
        n: 169.22  attr: {'dast': True, 't-at-school': 5}  rel: {'__at__': Site(home) }
        n: 109.99  attr: {'dast': True, 't-at-school': 6}  rel: {'__at__': Site(home) }
        n: 109.99  attr: {'dast': True, 't-at-school': 6}  rel: {'__at__': Site(home) }
        n:   33.0  attr: {'dast': True, 't-at-school': 7}  rel: {'__at__': Site(home) }
        n:   33.0  attr: {'dast': True, 't-at-school': 7}  rel: {'__at__': Site(home) }
        n:   5.08  attr: {'dast': True, 't-at-school': 8}  rel: {'__at__': Site(home) }
        n:   5.08  attr: {'dast': True, 't-at-school': 8}  rel: {'__at__': Site(home) }
```

References

[1] Grefenstette, J.J., Brown, S.T., Rosenfeld, R., Depasse, J., Stone, N.T., Cooley, P.C., Wheaton, W.D., Fyshe, A., Galloway, D.D., Sriram, A., Guclu, H., Abraham, T., Burke, D.S. (2013) FRED (A Framework for Reconstructing Epidemic Dynamics): An Open-Source Software System for Modeling Infectious Diseases and Control Strategies Using Census-Based Populations. *BMC Public Health, 13(1).*

[2] Knuth, D. (1974) Computer Programming as an Art. *Communications of the ACM, 17(12).*