Effects in RSiena on the C++ side

Materials for an online workshop

Tom A.B. Snijders

University of Oxford
University of Groningen

April, 2025

Overview of constructing effects

To construct a new effect, you have to do the following things.

1. Choose an `effectName` and a `shortName`.
2. Define the effect statistic and the change statistic.
3. Insert a new line in the code in `\data\allEffects.csv`.
4. Construct a new effect class in `\src\model\effects`,
   or add to an existing effect class.
5. Attach the new effect by appropriate lines in the `EffectFactory`.
6. If you constructed a new effect class,
   insert the name of the `cpp` file in `\src\sources.list`
   and the name of the `h` file in `\src\model\effects\allEffects.h`
7. Check that the effect runs properly, similarly as what was done in
   `https://github.com/stocnet/rsiena/blob/main/checkEffects.R`.

Points 4–6 are explained below.

Effects in RSiena

This is an overview of the construction of effects at the C++ side in RSiena .

It has to be combined with an overview of effects at the R side.

The C++ code of RSiena resides in the **`src`** directory of the source code
`https://github.com/stocnet/rsiena/blob/main/data`
Variable names in C++ – RSiena starting with the letter **`p`** are **`p`**ointers.

Recall:
Variables and functions in a class can be:

- ⇒ public: available to all classes that include the header file;
- ⇒ protected: available only to its descendants;
- ⇒ private: available only with the class (the **`.cpp`** file).
  Their names normally start with the letter **`l`** for **`l`**ocal.

Array indexing in C++ starts at 0.

Use of C++ in RSiena

The connection between the R and the C++ parts is made by
`R\initializeFRAN.r` and **`src\init.cpp`**

The C++ classes are defined in separate header files
`*\.h`
where the variables and functions are defined.
For almost all of them, the contents of these are specified in files
`*\.cpp`

Effects

The principle of inheritance is fundamental for the definition of effects.

There is a special directory **`src\model\effects`**.
Basic classes are **`EffectInfo`** defined in **`src\model`**,
and **`Effect`** defined in **`src\model\effects`**.

`EffectInfo` contains what is essential from `allEffects.csv`
to define the effect in C++ (the `shortName` is represented as `effectName`).

Almost all classes in **`src\model\effects`** are descendants of class **`Effect`**
(except those in **`src\model\effects\generic`**).

The construction of the effects object in RSiena
is based on `effects.r` on the R side,
and the **`EffectFactory`** on the C++ side.

In `EffectFactory.cpp`, each `shortName` (represented as `effectName`)
of an effect in the objective function is mentioned
and combined with an implementation using an effect class.

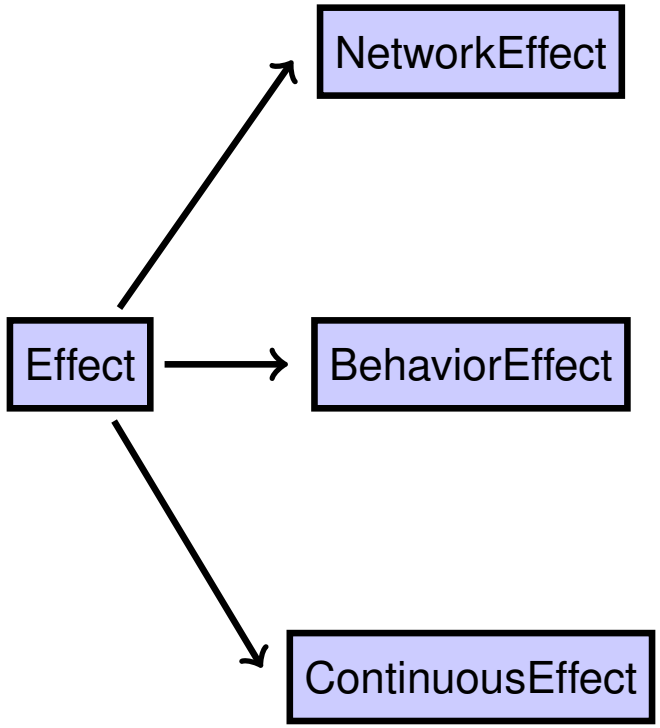
Direct descendants of `Effect`

The basic specification of effects is given in the final descendants of **`Effect`**.

For each effect, we need two things:

1. its contribution
   to the objective function
   (exception: `gmom`-type effects);
2. its estimation statistic for MoM
   (or for the GMoM).

(ML estimation is based only on the contribution to the objective function.)



Direct descendants
of **`Effect`**

Basic structure of network and behavior effects

The classes **`NetworkEffect`** and **`BehaviorEffect`** contain a lot.
Part of this are variables and functions that are
the raw material for creating effects.
Normally, you can take this for granted.
Sometimes, you may need to add to this.
Much of these raw materials are mentioned in these slides.

For the purpose of understanding effects and creating new ones,
the most important functions that you need to redefine
are specified in the following slides.

It is good to know that effects are calculated
for each actor separately, so there always will be one **`ego`**.

Making new effect classes or changing existing ones

When you wish to construct a new effect, you have a choice
between creating a new effect class for this purpose
(and then you have to add the name of the `.cpp` file to `sources.list`
and the name of the `.h` file to `\src\model\effects\allEffects.h`)
or modifying an existing effect class,
probably by the use of new internal parameters.

Another possibility is to define it as a generic network effect;
then you have to construct a new alter function class
(so you have to add the name of the `.cpp` file of this class to `sources.list`)
or use a new combination of existing alter functions.

For an interesting way to define several effects by one effect class using
a choice between outgoing and incoming tie iterators, you may look at
`DoubleDegreeBehaviorEffect.cpp`,
one effect class used for defining 6 effects.

Defining things for network effects

For specific network effects, i.e., descendants of **`NetworkEffect`**, the following virtual functions should be defined:

1. contribution to the objective function:
   `virtual double calculateContribution(int alter);`
   perhaps `virtual void preprocessEgo(int ego);`
2. estimation statistic for MoM:
   `virtual double egoStatistic();` or
   `virtual double tieStatistic();`

Other parts of the effect class are (of course) also important, such as its constructor (e.g., making available local variables like internal effect parameters) and **`initialize`**.

`preprocessEgo`

`preprocessEgo(ego)` is meant for gaining efficiency by calculating things that are not specific to alters,
both for network and for behavior effects.

It is a virtual function.
If you want to redefine it for a descendant of some effect class,
make sure to first use the **`preprocessEgo(ego)`** for the parent class.

See, e.g., **`OutOutDegreeAssortativityEffect`**.

Change statistic for network effects

The contribution to the evaluation function,

`virtual double calculateContribution(int alter)`

is the change statistic for effect k, i.e.,

$$\Delta_{kij}(x) = s_{ik}(x^{(+ij)}) - s_{ik}(x^{(-ij)}) ,$$

the difference of the evaluation function with and
the evaluation function without the tie $i \rightarrow j$.

Components of evaluation function for network effects

In **`NetworkEffect.cpp`**,
the virtual function **`egoStatistic`** is computed as

$$\text{egoStatistic}(i) = \sum_j x_{ij}\, \text{tieStatistic}(i,j)\ .$$

Note that both these functions are virtual.

It is necessary to redefine **`egoStatistic`** and/or **`tieStatistic`**.
Network effects with `interactionType = "ego"` or `"dyadic"`
can be used in interactions; for such effects the **`tieStatistics`** are used,
so these should be redefined for all such effects.

When you redefine both, then take care that the equation above is satisfied.

Endowment and creation effects

The endowment and creation effects will be computed by applying **`egoStatistic`** to the network of lost or (respectively) newly created ties;
only if this is to be replaced by something else,
should **`endowmentStatistic`** (or **`creationStatistic`**) be defined.

Helpful contents of class **NetworkEFfect**

For use in descendants of class **NetworkEFfect**,
the following is among what is made available:

```
Network * pNetwork();
NetworkLongitudinalData * pData();
```

`pNetwork()` is the current simulated network.
`pData()` is the network data set.

Helpful contents of class **Network**

... and through `pNetwork`, there is access, e.g., to:

`int n();` (number of actors)
`int m();` (number of 'alters')

```
TieIterator ties();
IncidentTieIterator inTies(int i);
IncidentTieIterator outTies(int i);
int inDegree(int i);
int outDegree(int i);
virtual bool isOneMode();
```

For the syntax of using the tie iterators,
see the examples in many of the effects.

Examples for network effects

As examples, first have a look at **ReciprocityEffect**,
which has change statistic

$$x_{ji} \ .$$

Then look at **IndegreePopularityEffect**, an effect class used for the effects **inPop**, **inPopSqrt**, and **inPop.c**.
The change statistics here are, respectively,

$$\sum_h x_{hj}, \quad \sqrt{\sum_h x_{hj}}, \quad \text{and} \quad \left(\sum_h x_{hj}\right) - d$$

(d is average indegree over all waves).

Here the constructor defines various internal parameters,
and the initialization calculates d.

Direct descendants of **NetworkEffect**

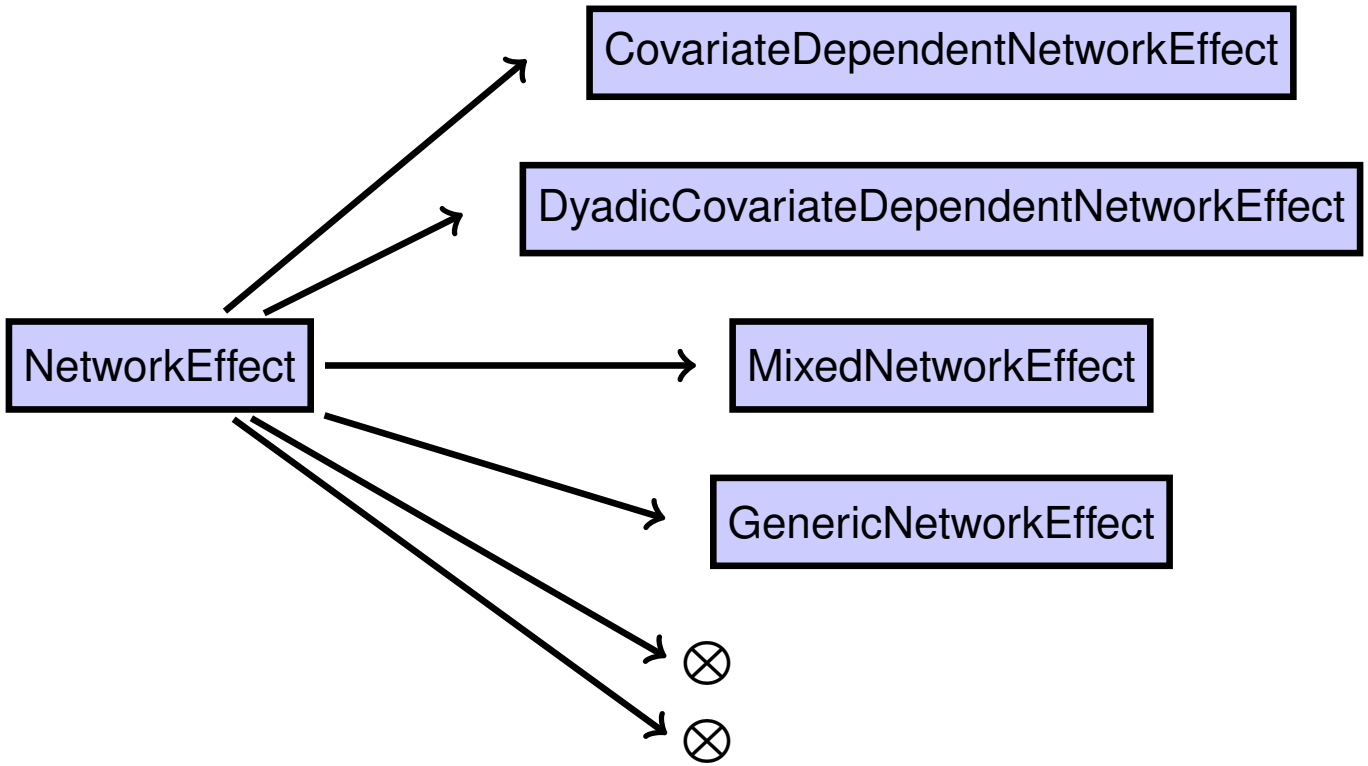
Many interesting network effects also depend
on monadic or dyadic covariates, or on other networks.

Co-dependence on behavior is treated just like
co-dependence on a monadic covariate.

The direct descendants of **NetworkEffect** are many network-only effects,
and further umbrella classes that implement dependence on other structures,
and have their own descendants implementing effects.
See the next page.

This also mentions a special umbrella class called
GenericNetworkEffect.

Direct descendants of **NetworkEffect**



Direct descendants of **NetworkEffect**

There are a lot of ⊗ classes, which
define specific network-only effects such as **ReciprocityEffect**.

The class **CovariateDependentNetworkEffect**

This class makes available characteristics of monadic covariates
and of behavioral variables for their effects on a dependent network.
It does not define a specific effect, but is an umbrella class.

Some of the protected functions in class
CovariateDependentNetworkEffect:

```
double value(int i);
bool missing(int i);
```

The function `value` gives the covariate value;
for behavioral variables, this is the centered value.

Recall that in RSiena, monadic covariates are centered in the R side,
and dyadic covariates are centered within C++.

The class DyadicCovariateDependentNetworkEffect

This class makes available characteristics of dyadic covariates for their effects on a dependent network.

Some protected functions in this class:

```
double value(int i, int j);
bool missing(int i, int j);
DyadicCovariateValueIterator rowValues(int i);
DyadicCovariateValueIterator columnValues(int j);
```

The function `value` gives the centered dyadic covariate value; if the dyadic covariate was given as 'non-centered' to RSiena, the non-centered covariate value.

Class **DyadicCovariateDependentNetworkEffect** (2)

```
DyadicCovariateValueIterator rowValues(int i);
DyadicCovariateValueIterator columnValues(int j);
```

These two iterators are given for improving
computational efficiency in the case of sparse dyadic covariates.
They will give the non-centered values of the dyadic covariates.

They are used for triadic effect classes such as **WWXClosureEffect**
implementing the `WWX` effect and its relatives.

The class **MixedNetworkEffect**

This class makes available characteristics of other networks
for their co-evolution effects on a dependent network.

Some protected functions in this class:

```
bool firstOutTieExists(int alter);
bool secondOutTieExists(int alter);
Network * pFirstNetwork();
Network * pSecondNetwork();
TwoNetworkCache * pTwoNetworkCache();
```

Similar functions for in-ties are not included.

Many effects for co-evolution of multiple networks
are defined not as descendants of this class, but as generic effects.

The class `GenericNetworkEffect`

The class **`GenericNetworkEffect`** may be used to specify an effect for a network variable x if its evaluation and endowment statistics can be conveniently expressed as

$$\sum_{j;j\neq i} x_{ij}\, f_{ij}(x, z)$$

and

$$\sum_{j;j\neq i} (1 - x_{ij})\, x_{ij}^{\text{obs}}(t_m)\, f_{ij}(x^{\text{obs}}(t_m), z)\ ,$$

respectively. This function $f_{ij}(x, z)$ as well as the change contribution $\Delta_{ij}(x, z)$ (tie statistic!) should be specified as instances of the **`AlterFunction`** class and passed as parameters when creating the effect.
Normally, they are almost identical, except that missingness of covariate values is taken into account for $f_{ij}(x, z)$.

`egoStatistic`s are not separately defined for this class.

Advantages of generic network effects

The point of generic network effects is that they allow
simpler and more flexible/reuseable construction of effects,
by using arguments of alter functions and composition of functions.

For function composition, there are available, e.g.,
`AbsDiffFunction`, **`ConditionalFunction`**,
`ConstantFunction`, **`ProductFunction`**,
`ReciprocalFunction`, and **`SumFunction`**.

Definition of `GenericNetworkEffect`

The class **`GenericNetworkEffect`** is defined as

```
GenericNetworkEffect(EffectInfo * pEffectInfo,
                     AlterFunction * pFunction);
GenericNetworkEffect(EffectInfo * pEffectInfo,
                     AlterFunction * pEffectFunction,
                     AlterFunction * pStatisticFunction);
```

The `pEffectFunction` is the change statistic $\Delta_{ij}(x, z)$;
the `pStatisticFunction` is the tie statistic $f_{ij}(x, z)$.
If the first form is used, they are the same.
Making them different can be useful, e.g.,
to take account of what are missing data.
When used for a GMoM statistic, the `pEffectFunction` is irrelevant.

Examples of **`GenericNetworkEffect`**

Whether the use of generic network effects always
results in simpler coding may be a matter of taste.

In **`effectFactory.cpp`**, you may look, e.g., at
effects `crprod`, `crprodRecip`, `crprodMutual`, and `crprod_gmm`;
and then at the **`AlterFunction`**s they use.

For more complicated examples, look at
`inPopX`, `inPopIntn` and `inPopIntnX`.

Effects `doubleOutAct` and `doubleInPop` are
implemented differently
(one with a generic effect, the other with a MixedNetworkEffect).
You could look at the differences.

Defining things for behavior effects

For descendants of **`BehaviorEffect`**,
the following virtual functions should be redefined:

1. contribution to the objective function:
   ```
   virtual double calculateChangeContribution
                  (int actor, int difference) ;
   ```
   perhaps `virtual void preprocessEgo(int ego);`
2. estimation statistic for MoM:
   ```
   virtual double egoStatistic
                  (int ego, double * currentValues).
   ```
   The `currentValues` will be the
   simulated values at the end of the period.

Furthermore, it may be necessary to adapt the constructor of the effect
to make available local variables (e.g., internal effect parameters)
and **`initialize`**.

Change statistic for behavior effects

The contribution to the evaluation function,

```
virtual double calculateChangeContribution
                    (int actor, int difference) ;
```

is the change statistic for effect k

when `difference` $= d$ is added to the behavior i.e.,

$$\Delta_{kid}(x, z) = s_{ik}(x, z + d) - s_{ik}(x, z) .$$

It needs to be defined for `difference=+1` and `difference=-1`,

and to be sure, for `difference=0`.

Some elements of the **Behavioreffect** class

The **Behavioreffect** class has, in its *protected* part, variables such as

- `n()` number of actors
- `value(int actor)` non-centered behavior
- `centeredValue(int actor)` centered behavior
- `initialValue(int actor)` non-centered behavior at start of the period
- `missing(int observation, int actor)` is the behavior value missing for this actor and observation

So using `this->` followed by any of these, in an effect of a descendant of class **Behavioreffect**, will be the corresponding characteristic of the behavior dependent variable.

Example for behavior effects

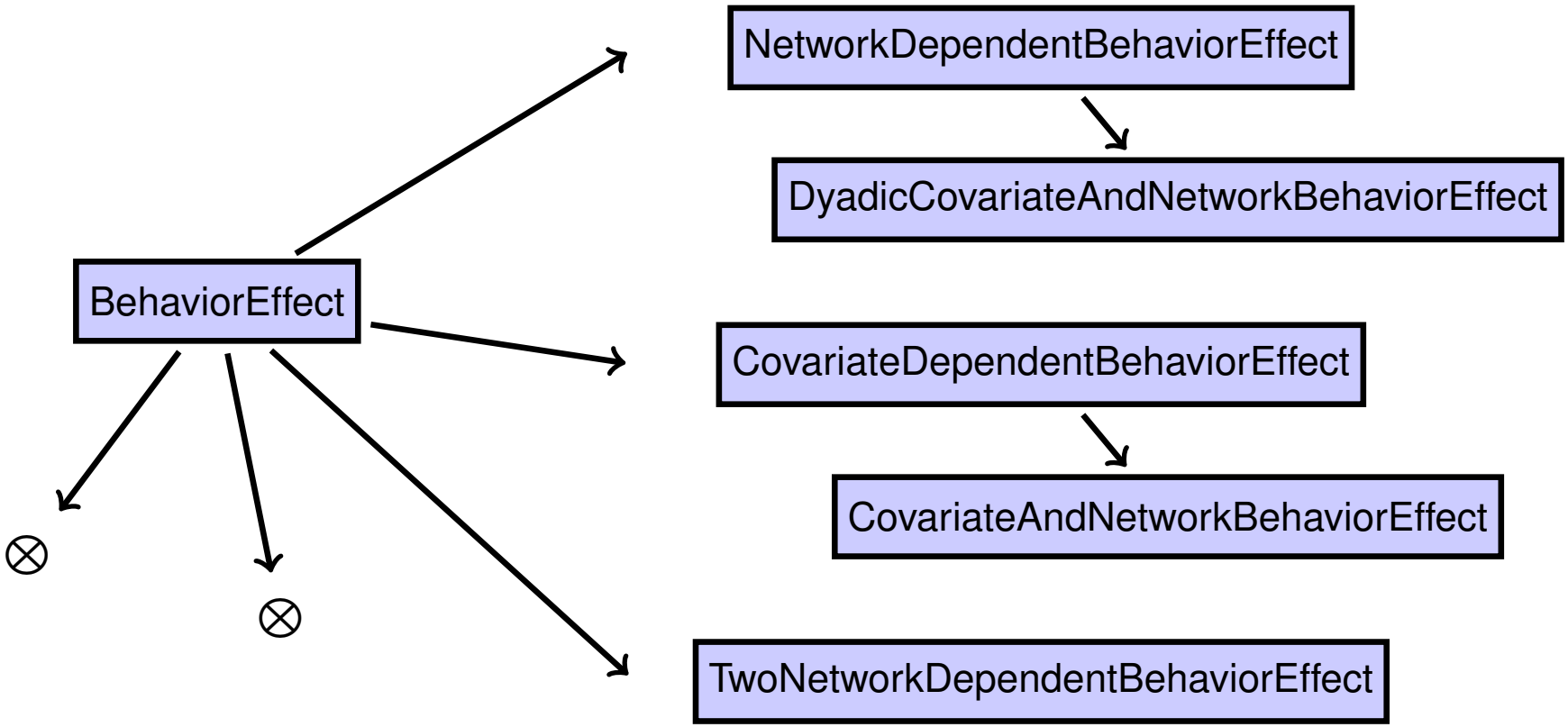
As a simple example of an effect implemented by a direct descendant, have a look at **QuadraticShapeEffect**, with change contribution

$$d * (2 * z_i + d) \ .$$

Most interesting behavior effects also depend on the network,
or on covariates.

The direct descendants of **Behavioreffect** are
a few behavior-only effects, and furthermore
umbrella classes that implement dependence on other structures,
and have their own descendants implementing effects.
This is shown on the following page.

Direct descendants of **BehaviorEffect**



Direct descendants of **NetworkEffect** and further umbrella classes

There are several $\otimes$ classes, defining specific behavior-only effects such as **LinearShapeEffect** and **AverageGroupEffect**.

The class **NetworkDependentBehaviorEffect**

This class makes available characteristics of networks for their co-evolution effects with behavior.

Some protected functions in this class:

```
Network * pNetwork();
double totalAlterValue(int i);
double totalInAlterValue(int i);
int numberAlterHigher(int i);
int numberAlterLower(int i);
int numberAlterEqual(int i);
int numberAlterHigherPop(int i);
int numberAlterLowerPop(int i);
int numberAlterEqualPop(int i);
```

Through `pNetwork`, also degrees, iterators, etc.

Direct descendants of **NetworkDependentBehaviorEffect**

This class has descendants implementing important effects, e.g., **AverageAlterEffect** and **OutdegreeEffect**.

There further is the umbrella class **DyadicCovariateAndNetworkBehaviorEffect**.

The class **DyadicCovariateAndNetworkBehaviorEffect**

This class makes available characteristics of dyadic covariates for their use in co-evolution of behavior and networks.

Some protected functions in this class:

```
double dycoValue(int i, int j);
bool missingDyCo(int i, int j);
DyadicCovariateValueIterator rowValues(int i);
DyadicCovariateValueIterator columnValues(int j);
```

This is similar to class **DyadicCovariateDependentNetworkEffect**, but care should be taken with the different names `dycoValue` and `missingDyCo`.

The class **CovariateDependentBehaviorEffect**

This class makes available characteristics of monadic covariates for their use in co-evolution of behavior and networks.

Some protected functions in this class:

```
double covariateMean();
double covariateValue(int i);
bool missingCovariate(int i, int observation);
bool missingCovariateEitherEnd(int i, int observation);
```

The class **CovariateAndNetworkBehaviorEffect**

This is a child of class **CovariateDependentBehaviorEffect**,
and adds some functions of the network.
It makes available characteristics of the network
in addition to those already in **CovariateDependentBehaviorEffect**.

(It would have been more efficient to make it a child of both
CovariateDependentBehaviorEffect and **NetworkDependentBehaviorEffect**....)

Some protected functions in this class:

```
Network * pNetwork();
bool missingDummy(int i);
bool missingInDummy(int i);
double averageAlterValue(int i);
double minimumAlterValue(int i);
double maximumAlterValue(int i);
double totalAlterValue(int i);
double averageInAlterValue(int i);
double totalInAlterValue(int i);
```

The class **TwoNetworkDependentBehaviorEffect**

This class makes available characteristics of two networks
for their use in co-evolution of behavior depending on two networks.

Some protected functions in this class:

```
Network * pFirstNetwork();
Network * pSecondNetwork();
double firstTotalAlterValue(int i);
double firstTotalInAlterValue(int i);
```

This is used only for the implementation of
the double degree effects `FFdeg`, etc.

Class **ContinuousEffect**

Continuous effects are used in the specification
of models for continuous behavioral dependent variables.

For specific continuous effects, i.e., descendants of **ContinuousEffect**,
the following virtual functions should be defined:

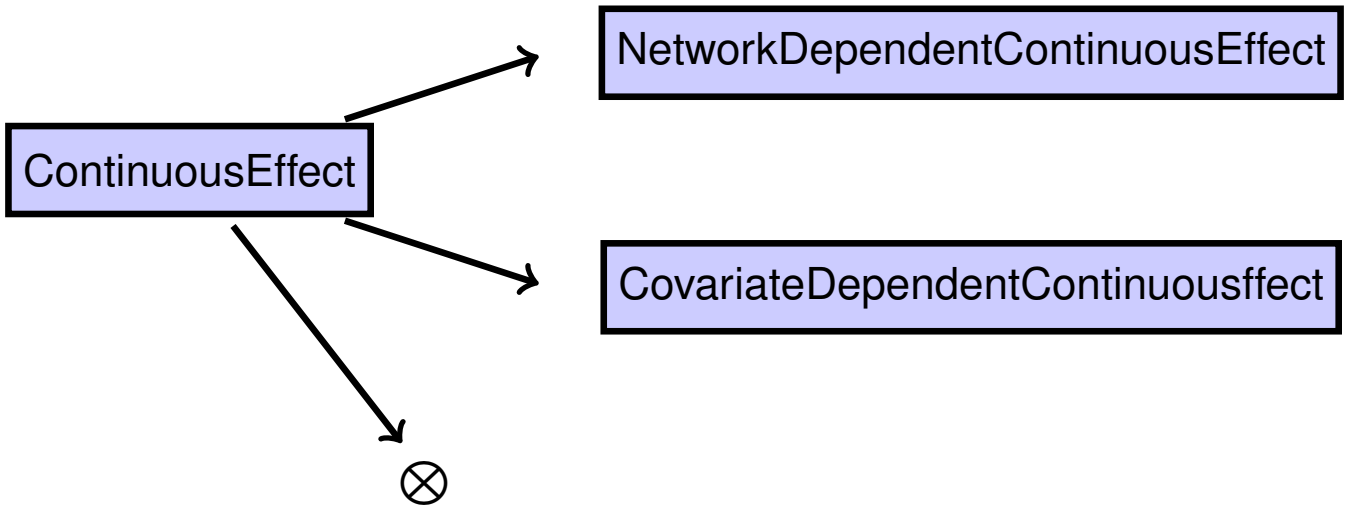
1. contribution to change in continuous behavior:
   ```
   virtual double calculateChangeContribution(int actor);
   ```
2. estimation statistic for MoM:
   ```
   virtual double egoStatistic(int ego,
                               double * currentValues)
   ```
   The `currentValues` will be the
   simulated values at the end of the period.

Protected functions in **ContinuousEffect**

This class makes available the following characteristics of the continuous behavior variable.

```
int n();
double value(int actor);
double centeredValue(int actor);
bool missing(int observation, int actor);
```

Direct descendants of `ContinuousEffect`



Direct descendants of `ContinuousEffect`

There is one ⊗ class: `InterceptEffect`.

The class **NetworkDependentContinuousEffect**

This class makes available characteristics of the network
for their effects on a continuous behavior variable.
It does not define a specific effect, but is an umbrella class.

The only protected function in class
CovariateDependentNetworkEffect:

```
Network * pNetwork();
```

Through this function, everything in the **Network** class is accessible.

The class **CovariateDependentContinuousEffect**

This class makes available characteristics of a covariate
for their effects on a continuous behavior variable.
It does not define a specific effect, but is an umbrella class.

The protected functions in class **CovariateDependentNetworkEffect**:

```
double covariateValue(int i);
bool missingCovariate(int i, int observation);
bool missingCovariateEitherEnd(int i, int observation);
```

Two examples

Example of a **NetworkEffect**: `avDeg` in

`https://github.com/stocnet/rsiena/blob/main/src/model/effects/AverageDegreeEffect.cpp`

Example of a **GenericNetworkEffect**:

Search for `outAct_ego` in

`https://github.com/stocnet/rsiena/blob/main/src/model/effects/EffectFactory.cpp`

and then

`https://github.com/stocnet/rsiena/blob/main/src/model/effects/generic/EgoOutDegreeFunction.cpp`

Checking the effect

Checking the C++ and R syntax is already incorporated in the package check.

What is left at the end is to check that the effect does what you want.

I must confess that usually I do this in a limited way:

1. Check that estimation of the model with the new effect converges (for all relevant data types: do not forget two-mode networks).
2. Check the evaluation statistic.

This does not explicitly include a check of the change statistic, which is because this is more difficult to get as a result of a function in the package.

Steps in the effect check

The effects are checked in
`https://github.com/stocnet/rsiena/blob/main/checkEffects.R`
This is in the github code, but it is mentioned in `.Rbuildignore`,
so it is not in the R code Please add your check at the end.
You can see in in `checkEffects.R` examples of how this was done.

First create a data set without missings with what is internally available,
and specify a model with the new effect (in all possible variations),
and estimate it. Check that the effect name reads well.

This should converge right away.
Request the `targets` of the result, and check those with what you can calculate 'manually' (see the examples in `checkEffects.R`).

If you have modified an existing effect, please also recheck the existing effect, to make sure that you did not accidentally damage it.

Wrap-up

To construct a new effect, you have to do the following things.

1. Choose an `effectName` and a `shortName`.
2. Define the effect statistic and the change statistic.
3. Insert a new line in the code in `\data\allEffects.csv`.
4. Construct a new effect class in `\src\model\effects`,
   or add to an existing effect class.
5. Attach the new effect by appropriate lines in the `EffectFactory`.
6. If you constructed a new effect class,
   insert the name of the `cpp` file in `\src\sources.list`
   and the name of the `h` file in `\src\model\effects\allEffects.h`
7. Check that the effect runs properly, similarly as what was done in
   `https://github.com/stocnet/rsiena/blob/main/checkEffects.R`.

And if you want to add your effect to RSiena at GitHub:

make an entry for Chapter 12 in the Siena manual, and:

make a pull request, or raise an issue,
or send a message to `RSiena@groups.io`,
or send an email message to Christian or Tom.


The horserace of Siena,
Il palio was established in 1681