Title

stata.com

gsem path notation extensions — Command syntax for path diagrams

Description Syntax Options Remarks and examples Also see

Description

This entry concerns `gsem` only.

The command syntax for describing generalized SEMs is fully specified by *paths*, `covariance()`, `variance()`, `covstructure()`, and `means()`; see [SEM] **sem and gsem path notation** and [SEM] **sem and gsem option covstructure( )**.

With `gsem`, the notation is extended to allow for generalized linear response variables, multilevel latent variables, categorical latent variables, and comparisons of groups. That is the subject of this entry.

Syntax

`gsem` *paths* ... [`, covariance() variance() means() group() lclass()`]

`gsem` *paths* ... [`, covstructure() means() group() lclass()`]

paths specifies the direct paths between the variables of your model.

The model to be fit is fully described by *paths*, `covariance()`, `variance()`, `covstructure()`, and `means()`.

The syntax of these elements is modified when the `group()` or `lclass()` option is specified.

Options

`covariance()`, `variance()`, and `means()` are described in [SEM] **sem and gsem path notation**.

`covstructure()` is described in [SEM] **sem and gsem option covstructure( )**.

`group(`*varname*`)` allows models specified with *paths*, `covariance()`, `variance()`, `covstructure()`, and `means()` to be automatically interacted with the groups defined by *varname*; see [SEM] **intro 6**. The syntax of *paths* and the arguments of `covariance()`, `variance()`, `covstructure()`, and `means()` gain an extra syntactical piece when `group()` is specified.

`lclass()` allows models specified with *paths*, `covariance()`, `variance()`, and `covstructure()` to be automatically interacted with categorical latent variables; see [SEM] **intro 2**. The syntax of *paths* and the arguments of `covariance()`, `variance()`, and `covstructure()` gain an extra syntactical piece when `lclass()` is specified.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Specifying family and link

gsem fits not only linear models but also generalized linear models. There is a set of options for specifying the specific model to be fit. These options are known as family-and-link options, which include family() and link(), but those options are seldom used in favor of other family-and-link shorthand options such as logit, which means family(bernoulli) and link(logit). These options are explained in [SEM] **gsem family-and-link options**.

In the command language, you can specify these options among the shared options at the end of a gsem command:

```
. gsem ..., ... logit ...
```

That is convenient but only if all the equations in the model are using the same specific response function. Many models include multiple equations with each using a different response function.

You can specify any of the family-and-link options within paths. For instance, typing

```
. gsem (y <- x1 x2), logit
```

has the same effect as typing

```
. gsem (y <- x1 x2, logit)
```

Thus you can type

```
. gsem (y1 <- x1 L, logit) (y2 <- x2 L, poisson) ..., ...
```

The y1 equation would be logit, and the y2 equation would be Poisson. If you wanted y2 to be linear regression, you could type

```
. gsem (y1 <- x1 L, logit) (y2 <- x2 L, regress) ..., ...
```

or you could be silent and let y2 default to linear regression,

```
. gsem (y1 <- x1 L, logit) (y2 <- x2 L) ..., ...
```

Specifying multilevel nested latent variables

Latent variables are indicated by a name in which at least the first letter is capitalized. This generic form of the name is often written *Lname*.

In regular latent variables, which we will call level-1 latent variables, the unobserved values vary observation by observation. Level-1 latent variables are the more common kinds of latent variables.

gsem allows higher-level latent variables as well as the level-1 variables. Let's consider three-level data: students at the observational level, teachers at the second level, and schools at the third. In these data, each observation is a student. We have data on students nested within teachers nested within schools.

Let's assume that we correspondingly have three identification (ID) variables. We number the following list with the nesting level of the data:

3. Variable school contains a school ID number. If two observations have the same value of school, then both of those students attended the same school.
2. Variable teacher contains a teacher ID number, or it contains a teacher-within-school ID number. That is, we do not care whether different schools assigned teachers the same ID number. It will be sufficient for us that the ID number is unique within school.

1. Variable `student` contains a student ID number, or it contains a student-within-school ID number, or even a student-within-teacher-within-school ID number. That is, we do not care whether different observations have the same student ID as long as they have different teacher IDs or different school IDs.

Here is how you write latent variable names at each level of the model:

3. Level 3 is the school level. Latent variables are written as

   Lname`[school]`

   An example would be `SchQuality[school]`.

   The unobserved values of *Lname*`[school]` vary across schools and are constant within school.

   If *Lname*`[school]` is endogenous, its error variable is `e.`*Lname*`[school]`.

   You must refer to *Lname*`[school]` without omitting the `[school]` part. *Lname* by itself looks like another latent variable to `gsem`.

2. Level 2 is the teacher-within-school level. Latent variables are written as

   Lname`[school>teacher]` or
   Lname`[teacher<school]`

   An example would be `TeachQuality[school>teacher]` or
   `TeachQuality[teacher<school]`.

   To `gsem`, *Lname*`[school>teacher]` and *Lname*`[teacher<school]` mean the same thing. You can even refer to `TeachQuality[school>teacher]` in one place and refer to `TeachQuality[teacher<school]` in another, and there will be no confusion.

   The unobserved values of *Lname*`[school>teacher]` vary across schools and teachers, and they are constant within teacher.

   If *Lname*`[school>teacher]` is endogenous, its error variable is
   `e.`*Lname*`[school>teacher]` or, equivalently, `e.`*Lname*`[teacher<school]`.

1. Level 1 is the student or observational level. Latent variables are written as

   Lname`[school>teacher>student]` or
   Lname`[student<teacher<school]` or
   Lname

   Everybody just writes *Lname*. These are the latent variables that correspond to the latent variables that `sem` provides. Unobserved values within the latent variable vary observation by observation.

   If *Lname* is endogenous, its error variable is `e.`*Lname*.

You can use multilevel latent variables in paths and options just as you would use any other latent variable; see [SEM] **sem and gsem path notation**. Remember, however, that you must type out the full name of all but the first-level latent variables. You type, for instance, `SchQual[school>teacher]`. There is a real tendency to type just `SchQual` when the name is unique.

Changing the subject, we see that the names by which effects are referred to are a function of the top level. We just discussed a three-level model. The three levels of the model were

(3) `school`
(2) `school>teacher`
(1) `school>teacher>student`

If we had a two-level model, the levels would be

(2) `teacher`
(1) `teacher>student`

Thus, if we had started with a two-level model and then wanted to add a third, higher level onto it, latent variables that were previously referred to as, say, `TeachQual[teacher]` would now be referred to as `TeachQual[school>teacher]`.

Specifying multilevel crossed latent variables

In our previous example, we had a three-level nested model in which student was nested within teacher, which was nested within school.

Let's consider data on employees that also have the characteristics of working in an occupation and working in an industry. These variables are not nested. Just as before, we will assume we have variable `employee` containing an employee ID and variables `industry` and `occupation`. The latent variables associated with this model could be the following:

Level	Latent-variable name
occupation	*Lname*`[occupation]`
industry	*Lname*`[industry]`
employee (observational)	*Lname*

Specifying paths for a specific group

The `group(`*varname*`)` option,

```
. gsem ..., ... group(varname)
```

specifies that the model be fit separately for the different values of *varname*. *varname* might be `sex`, and then the model would be fit separately for males and females, or *varname* might be something else and perhaps take on more than two values.

Whatever *varname* is, `group(`*varname*`)` defaults to letting some of the path coefficients, covariances, variances, and means of your model vary across the groups and constraining others to be equal. Which parameters vary and which are constrained is described in [SEM] **gsem group options**, but that is a minor detail right now.

In what follows, we will assume that *varname* is `mygrp` and takes on three values. Those values are 1, 2, and 3, but they could just as well be 2, 9, and 12.

Consider typing

```
. gsem ..., ...
```

and typing

```
. gsem ..., ... group(mygrp)
```

Whatever *paths*, `covariance()`, `variance()`, `covstructure()`, and `means()` are that describe the model, there are now three times as many parameters because each group has its own unique set. In fact, when you give the second command, you are not merely asking for three times the parameters, you are specifying three models, one for each group! In this case, you specified the same model three times without knowing it.

You can vary the model specified across groups.

1. Let's write the model you wish to fit as

```
. gsem (a) (b) (c), cov(d) cov(e) var(f)
```

where a, b, ..., f stand for what you type. In this generic example, we have two `cov()` options just because multiple `cov()` options often occur in real models. When you type

```
. gsem (a) (b) (c), cov(d) cov(e) var(f) group(mygrp)
```

results are as if you typed

```
. gsem (1: a) (2: a) (3: a)                                  ///
       (1: b) (2: b) (3: b)                                  ///
       (1: c) (2: c) (3: c),                                 ///
                cov(1: d) cov(2: d) cov(3: d)                ///
                cov(1: e) cov(2: e) cov(3: e)                ///
                var(1: f) cov(2: f) cov(3: f)  group(mygrp)
```

The `1:`, `2:`, and `3:` identify the groups for which paths, covariances, or variances are being added, modified, or constrained.

If `mygrp` contained the unique values 5, 8, and 10 instead of 1, 2, and 3, then `5:` would appear in place of `1:`, `8:` would appear in place of `2:`, and `10:` would appear in place of `3:`.

2. Consider the model

```
. gsem (y <- x) (b) (c), cov(d) cov(e) var(f) group(mygrp)
```

The default `ginvariant()` option constrains all intercepts, coefficients, and loadings to be the same across all groups. If you wanted to constrain the path coefficient `(y <- x)` to be the same across all three groups, but let all other parameters be group specific, you could type

```
. gsem (y <- x@c1) (b) (c), cov(d) cov(e) var(f) group(mygrp) ginvariant(none)
```

This works because the expansion of `(y <- x@c1)` is

```
(1: y <- x@c1) (2: y <- x@c1) (3: y <- x@c1)
```

See item 12 in [SEM] **sem and gsem path notation** for more examples of specifying constraints.

3. Consider the model

```
. gsem (y <- x) (b) (c), cov(d) cov(e) var(f) group(mygrp)
```

If you wanted to constrain the path coefficient `(y <- x)` to be the same in groups 2 and 3, you could type

```
. gsem (1: y <- x) (2: y <- x@c1) (3: y <- x@c1) (b) (c),      ///
                                cov(d) cov(e) var(f) group(mygrp)
```

The default `ginvariant()` option still constrains the other coefficients to be equal to each other, but not necessarily equal to the coefficient in groups 2 and 3. In our example, `mygrp` has 3 levels, so the above effectively removed the default constraint on the group 1 coefficient.

4. Instead of following item 3, you could type

```
. gsem (y <- x) (2: y <- x@c1) (3: y <- x@c1) (b) (c),        ///
                                cov(d) cov(e) var(f) group(mygrp)
```

The part `(y <- x) (2: y <- x@c1) (3: y <- x@c1)` expands to

```
(1: y <- x) (2: y <- x) (3: y <- x) (2: y <- x@c1) (3: y <- x@c1)
```

and thus the path is defined twice for group 2 and twice for group 3. When a path is defined more than once, the definitions are combined. In this case, the second definition adds more information, so the result is as if you typed

```
(1: y <- x) (2: y <- x@c1) (3: y <- x@c1)
```

5. Instead of following item 3 or item 4, you could type

```
. gsem (y <- x@c1) (1: y <- x@c2) (b) (c),      ///
                                cov(d) cov(e) var(f) group(mygrp)
```

The part `(y <- x@c1) (1: y <- x@c2)` expands to

```
(1: y <- x@c1)  (2: y <- x@c1)  (3: y <- x@c1)  (1: y <- x@c2)
```

When results are combined from repeated definitions, then definitions that appear later take precedence. In this case, results are as if the expansion read

```
(1: y <- x@c2) (2: y <- x@c1)  (3: y <- x@c1)
```

Thus coefficients for groups 2 and 3 are constrained. The group-1 coefficient is constrained to `c2`. If `c2` appears nowhere else in the model specification, then results are as if the path for group 1 were unconstrained.

6. Instead of following item 3, item 4, or item 5, you could not type

```
. gsem (y <- x@c1) (1: y <- x) (b) (c),                ///
                                cov(d) cov(e) var(f) group(mygrp)
```

The expansion of `(y <- x@c1) (1: y <- x)` reads

```
(1: y <- x@c1)  (2: y <- x@c1)  (3: y <- x@c1)  (1: y <- x)
```

and you might think that `1: y <- x` would replace `1: y <- x@c1`. Information, however, is combined, and even though precedence is given to information appearing later, silence does not count as information. Thus the expanded and reduced specification reads the same as if `1: y <- x` was never specified:

```
(1: y <- x@c1)  (2: y <- x@c1)  (3: y <- x@c1)
```

7. Items 1–6, stated in terms of *paths*, apply equally to what is typed inside the `means()`, `variance()`, `covariance()`, and `covstructure()` options. For instance, if you typed

```
. gsem (a) (b) (c), var(e.y@c1) group(mygrp)
```

then you are constraining the variance to be equal across all three groups.

If you wanted to constrain the variance to be equal in groups 2 and 3, you could type

```
. gsem (a) (b) (c), var(e.y) var(2: e.y@c1) var(3: e.y@c1), group(mygrp)
```

You could omit typing `var(e.y)` because it is implied. Alternatively, you could type

```
. gsem (a) (b) (c), var(e.y@c1) var(1: e.y@c2) group(mygrp)
```

You could not type

```
. gsem (a) (b) (c), var(e.y@c1) var(1: e.y) group(mygrp)
```

because silence does not count as information when specifications are combined.

Similarly, if you typed

```
. gsem (a) (b) (c), cov(e.y1*e.y2@c1) group(mygrp)
```

then you are constraining the covariance to be equal across all groups. If you wanted to constrain the covariance to be equal in groups 2 and 3, you could type

```
. gsem (a) (b) (c), cov(e.y1*e.y2)                                  ///
                    cov(2: e.y1*e.y2@c1) cov(3: e.y1*e.y2@c1) ///
                    group(mygrp)
```

You could not omit `cov(e.y1*e.y2)` because it is not assumed. By default, error variables are assumed to be uncorrelated. Omitting the option would constrain the covariance to be 0 in group 1 and to be equal in groups 2 and 3.

Alternatively, you could type

```
. gsem (a) (b) (c), cov(e.y1*e.y2@c1)                    ///
                    cov(1: e.y1*e.y2@c2)                  ///
                    group(mygrp)
```

8. In the examples above, we have referred to the groups with their numeric values, 1, 2, and 3. Had the values been 5, 8, and 10, then we would have used those values.

   If the group variable `mygrp` has a value label, you can use the label to refer to the group. For instance, imagine `mygrp` is labeled as follows:

   ```
   . label define grpvals 1 Male  2 Female  3 "Unknown sex"
   . label values mygrp grpvals
   ```

   We could type

   ```
   . gsem (y <- x) (Female: y <- x@c1) (Unknown sex: y <- x@c1) ..., ...
   ```

   or we could type

   ```
   . gsem (y <- x) (2: y <- x@c1) (3: y <- x@c1) ..., ...
   ```

Specifying paths for a specific latent class

The `lclass()` option in

```
. gsem ..., ... lclass(C 3)
```

specifies that the model be fit separately for each of 3 classes of the categorical latent variable `C`. The classes of `C` are 1, 2, and 3. `gsem` allows for more than one categorical latent variable to be specified by allowing more than one `lclass()` option. The two `lclass()` options in

```
. gsem ..., ... lclass(C 3) lclass(D 2)
```

specify that the model be fit separately for each of the 6 latent classes defined by the interaction between the categorical latent variables `C` and `D`.

Whatever the number of classes and latent variables, `gsem` defaults to letting some of the path coefficients, covariances, and variances of your model to vary across the latent classes and constraining others to be equal. Which parameters vary and which are constrained is described in [SEM] **gsem lclass options**, but that is a minor detail right now.

Consider typing

```
. gsem ..., ...
```

then typing

```
. gsem ..., ... lclass(C 3)
```

Whatever *paths*, `covariance()`, `variance()`, and `covstructure()` are that describe the model, there are now three times as many parameters because each latent class has its own unique set. In fact, when you give the second command, you are not merely asking for three times the parameters, you are specifying three models, one for each latent class, and a multinomial model for latent class probabilities.

You can vary the model specified across latent classes.

1. Let's write the model you wish to fit as

   ```
   . gsem (a) (b) (c), cov(d) cov(e) var(f)
   ```

   where a, b, ..., f stand for what you type. In this generic example, we have two `cov()` options just because multiple `cov()` options can occur in real models. When you type

   ```
   . gsem (a) (b) (c), cov(d) cov(e) var(f) lclass(C 3)
   ```

   results are as if you typed

   ```
   . gsem (1: a) (2: a) (3: a)                                  ///
          (1: b) (2: b) (3: b)                                  ///
          (1: c) (2: c) (3: c)                                  ///
          (1.C <- _cons@0)                                      ///
          (2.C <- _cons)                                        ///
          (3.C <- _cons),                                       ///
                   cov(1: d) cov(2: d) cov(3: d)                ///
                   cov(1: e) cov(2: e) cov(3: e)                ///
                   var(1: f) cov(2: f) cov(3: f)                ///
                   lclass(C 3)
   ```

   The `1:`, `2:`, and `3:` identify the latent classes for which paths, covariances, or variances are being added, modified, or constrained.

   The paths to `1.C`, `2.C`, and `3.C` identify linear predictions for the multinomial probabilities for the categorical latent variable `C`.

2. Consider the model

   ```
   . gsem (y <- x) (b) (c), cov(d) cov(e) var(f) lclass(C 3)
   ```

   If you wanted to constrain the path coefficient `(y <- x)` to be the same across all three latent classes, you could type

   ```
   . gsem (y <- x@c1) (b) (c), cov(d) cov(e) var(f) lclass(C 3)
   ```

   This works because the expansion of `(y <- x@c1)` is

   ```
   (1: y <- x@c1) (2: y <- x@c1) (3: y <- x@c1)
   ```

   See item 12 in [SEM] **sem and gsem path notation** for more examples of specifying constraints.

3. Consider the model

   ```
   . gsem (y <- x) (b) (c), cov(d) cov(e) var(f) lclass(C 3)
   ```

If you wanted to constrain the path coefficient `(y <- x)` to be the same in latent classes 2 and 3, you could type

```
. gsem (1: y <- x) (2: y <- x@c1) (3: y <- x@c1) (b) (c),      ///
                        cov(d) cov(e) var(f) lclass(C 3)
```

4. Instead of following item 3, you could type

```
. gsem (y <- x) (2: y <- x@c1) (3: y <- x@c1) (b) (c),        ///
                        cov(d) cov(e) var(f) lclass(C 3)
```

The part `(y <- x) (2: y <- x@c1) (3: y <- x@c1)` expands to

```
(1: y <- x) (2: y <- x) (3: y <- x) (2: y <- x@c1) (3: y <- x@c1)
```

and thus the path is defined twice for latent class 2 and twice for latent class 3. When a path is defined more than once, the definitions are combined. In this case, the second definition adds more information, so the result is as if you typed

```
(1: y <- x) (2: y <- x@c1) (3: y <- x@c1)
```

5. Instead of following item 3 or item 4, you could type

```
. gsem (y <- x@c1) (1: y <- x@c2) (b) (c),      ///
                        cov(d) cov(e) var(f) lclass(C 3)
```

The part `(y <- x@c1) (1: y <- x@c2)` expands to

```
(1: y <- x@c1)  (2: y <- x@c1)  (3: y <- x@c1)  (1: y <- x@c2)
```

When results are combined from repeated definitions, then definitions that appear later take precedence. In this case, results are as if the expansion read

```
(1: y <- x@c2) (2: y <- x@c1)  (3: y <- x@c1)
```

Thus coefficients for latent classes 2 and 3 are constrained. The first latent class coefficient is constrained to `c2`. If `c2` appears nowhere else in the model specification, then results are as if the path for the first latent class were unconstrained.

6. Instead of following item 3, item 4, or item 5, you could not type

```
. gsem (y <- x@c1) (1: y <- x) (b) (c),                ///
                        cov(d) cov(e) var(f) lclass(C 3)
```

The expansion of `(y <- x@c1) (1: y <- x)` reads

```
(1: y <- x@c1)  (2: y <- x@c1)  (3: y <- x@c1)  (1: y <- x)
```

and you might think that `1: y <- x` would replace `1: y <- x@c1`. Information, however, is combined, and even though precedence is given to information appearing later, silence does not count as information. Thus the expanded and reduced specification reads the same as if `1: y <- x` was never specified:

```
(1: y <- x@c1)  (2: y <- x@c1)  (3: y <- x@c1)
```

7. Items 1–6, stated in terms of *paths*, apply equally to what is typed inside the `variance()`, `covariance()`, and `covstructure()` options. For instance, if you typed

```
. gsem (a) (b) (c), var(e.y@c1) lclass(C 3) lcinvariant(none)
```

then you are constraining the error variance of `y` to be equal across all three latent classes. Without the `lcinvariant(none)` option, `gsem` would constrain all error variances to be equal across all groups.

If you wanted to constrain the variance to be equal in latent classes 2 and 3, you could type

```
. gsem (a) (b) (c), var(e.y) var(2: e.y@c1) var(3: e.y@c1), lclass(C 3)
```

You could omit typing `var(e.y)` because it is implied. Alternatively, you could type

```
. gsem (a) (b) (c), var(e.y@c1) var(1: e.y@c2) lclass(C 3)
```

You could not type

```
. gsem (a) (b) (c), var(e.y@c1) var(1: e.y) lclass(C 3)
```

because silence does not count as information when specifications are combined.

Similarly, if you typed

```
. gsem (a) (b) (c), cov(e.y1*e.y2@c1) lclass(C 3)
```

then you are constraining the covariance to be equal across all latent classes. If you wanted to constrain the covariance to be equal in latent classes 2 and 3, you could type

```
. gsem (a) (b) (c), cov(e.y1*e.y2)                                  ///
                    cov(2: e.y1*e.y2@c1) cov(3: e.y1*e.y2@c1) ///
                    lclass(C 3)
```

You could not omit `cov(e.y1*e.y2)` because it is not assumed. By default, error variables are assumed to be uncorrelated. Omitting the option would constrain the covariance to be 0 in latent class 1 and to be equal in latent classes 2 and 3.

Alternatively, you could type

```
. gsem (a) (b) (c), cov(e.y1*e.y2@c1) cov(1: e.y1*e.y2@c2) lclass(C 3)
```

8. The above examples focused on models with a single categorical latent variable. The same rules apply for models with two or more categorical latent variables, with the added requirement that the latent class prefix is specified using factor-variable notation.

   For example, suppose your model is

   ```
   . gsem (a), cov(b) var(c) lclass(C 3) lclass(D 2)
   ```

   The result is as if you typed

   ```
   . gsem (1.C#1.D: a) (1.C#2.D: a)                          ///
          (2.C#1.D: a) (2.C#2.D: a)                          ///
          (3.C#1.D: a) (3.C#2.D: a)                          ///
          (1.C <- _cons@0) (2.C <- _cons) (3.C <- _cons)     ///
          (1.D <- _cons@0) (2.D <- _cons),                   ///
                   cov(1.C#1.D: b) cov(1.C#2.D: b)           ///
                   cov(2.C#1.D: b) cov(2.C#2.D: b)           ///
                   cov(3.C#1.D: b) cov(3.C#2.D: b)           ///
                   var(1.C#1.D: c) var(1.C#2.D: c)           ///
                   var(2.C#1.D: c) var(2.C#2.D: c)           ///
                   var(3.C#1.D: c) var(3.C#2.D: c)           ///
                   lclass(C 3) lclass(D 2)
   ```

   Each of the *paths*, `variance()`, `covariance()`, and `covstructure()` options is now targeted to a specific latent class.

9. From the above example, we see that the multinomial probabilities for the latent classes are modeled as if `C` and `D` were independent. For example, the linear prediction for the latent class `3.C#2.D` is composed of the intercept for `3.C` plus the intercept for `2.D`.

   Add the path `3.C#2.D <- _cons` to the above model specification, and the linear prediction for the latent class `3.C#2.D` will be composed of the intercept for `3.C`, plus the intercept for `2.D`, plus the new specified intercept for `3.C#2.D`. With this new intercept, `C` and `D` are no longer modeled independently.

10. Consider the model

```
. gsem (y <- x), lclass(C 3) lclass(D 2)
```

If you wanted to constrain the path coefficient `(y <- x)` to be the same in all latent classes with `1.C`, then add the path `(1.C: y <- x@c1)` so that the model is

```
. gsem (y <- x) (1.C: y <- x@c1), lclass(C 3) lclass(D 2)
```

This essentially expands to

```
. gsem (1.C#1.D: y <- x@c1) (1.C#2.D: y <- x@c1)         ///
       (2.C#1.D: y <- x)    (2.C#2.D: y <- x)            ///
       (3.C#1.D: y <- x)    (3.C#2.D: y <- x)            ///
       (1.C <- _cons@0) (2.C <- _cons) (3.C <- _cons)    ///
       (1.D <- _cons@0) (2.D <- _cons),                  ///
                lclass(C 3) lclass(D 2)
```

This shortcut syntax works similarly for options `covariance()`, `variance()`, and `covstructure()`.

Specifying paths for a specific group and latent class

`gsem` allows models where both option `group()` and option `lclass()` are specified. Building on the examples from the above sections, the model

```
. gsem ..., ... group(mygrp) lclass(D 2)
```

specifies that the model with categorical latent variable `D` be fit separately for each level of `mygrp`. The number of model parameters multiplies when options `group()` and `lclass()` are specified.

For example, the model

```
. gsem (y <- x), group(mygrp) lclass(D 2)
```

with `mygrp` having levels 1, 2, and 3, is equivalent to

```
. gsem (1.mygrp#1.D: y <- x) (1.mygrp#2.D: y <- x)          ///
       (2.mygrp#1.D: y <- x) (2.mygrp#2.D: y <- x)          ///
       (3.mygrp#1.D: y <- x) (3.mygrp#2.D: y <- x)          ///
       (1.mygrp: 1.D <- _cons@0) (1.mygrp: 2.D <- _cons)    ///
       (2.mygrp: 1.D <- _cons@0) (2.mygrp: 2.D <- _cons)    ///
       (3.mygrp: 1.D <- _cons@0) (3.mygrp: 2.D <- _cons),   ///
                group(mygrp) lclass(D 2)
```

The path `(y <- x)` expands to 6 paths, one for each group and latent class combination. The number of covariances and variances is similarly multiplied by 6. The linear predictions for `D` expand by the number of group levels.

Also see

[SEM] **gsem** — Generalized structural equation model estimation command

[SEM] **gsem group options** — Fitting models on different groups

[SEM] **gsem lclass options** — Fitting models with latent classes

[SEM] **sem and gsem path notation** — Command syntax for path diagrams

[SEM] **intro 2** — Learning the language: Path diagrams and command language