# “FoldArchitect” or “BuildDeNovoBackboneMover” usage and examples

## Preface

If anything doesn’t work as you would expect it to, please contact Tom at tlinsky@gmail.com or estrauch@uga.edu.

## Concept

The BuildDeNovoBackboneMover has several components that work together to create a de novo backbone. The components uses are an architect, which designs the fold; a pose folder, which describes how the folding process should occur; a perturber, which tweaks the designs made by the architect to find parameters (e.g. length, register shift, ABEGO, etc) which correctly build the fold; PreFoldMovers, which are movers run immediately before folding; PostFoldMovers, which are movers run immediately after folding; and Filters, which tell the BuildDeNovoBackboneMover whether a fold has been successfully created. The BuildDeNovoBackboneMover utilizes a “divide-and-conquer” strategy, where the full fold is intelligently divided into subsegments. When one subsegment is folded properly, the mover gradually adds and and folds additional subsegments. This vastly improves the time required to generate poses with perfect secondary structure, and perhaps more importantly, allows sampling of different lenghts/abegos/other parameters DURING the process of building a pose. Therefore, one BuildDeNovoBackboneMover can easily diverse folds that mimic and even exceed the capability of 10,000 parallel runs using static BluePrint files.

## How it works

The BuildDeNovoBackboneMover will make one or more attempts to create a backbone that passes all user-specified filters. If a folding attempt fails, it will retry until an iterations limit is reached.

### Folding attempt

The BuildDeNovoBackboneMover uses four stages during an attempt build a backbone:

1. "Architects" design a "StructureData" object, which is an enhanced blueprint. Architects choose loop types, secondary structure lengths, etc from the set of values specified by the user.
2. "ExtendedChainPoseBuilder" (hidden to users) uses the StructureData generated by the architects to build a starting structure. Any new residues are created in the extended (phi/psi/omega = 180) conformation.

3. The pose is converted to centroid mode.
4. A customizable list of "Pre-Fold Movers" are run on the pose. The user-specified pre-fold movers must be compatible with centroid poses.
5. A "PoseFolder" generates a folded conformation. The RemodelLoopMoverPoseFolder uses fragment assembly to fold the backbone.
6. A customizable list of "Post-Fold Movers" are run on the pose. All post-fold movers must be compatible with centroid poses.
7. The pose is converted to full-atom mode.
8. A user-specified list of filters is run on the pose. In order for a folding attempt to be successful, the pose must pass all user-specified filters.

## Divide And Conquer

The BuildDeNovoBackboneMover uses a "divide-and-conqueor" algorithm which I designed when I joined the Baker Lab and was still building de novo structures using the old BlueprintBDR. Given a per-residue probability of correctly folding (Pres), the probability of folding an entire N-residue protein correctly is Pres^N. Although a 30-residue peptide may fold correctly 50% of the time, it may take thousands of iterations to get one 100-residue protein to fold. The key idea behind the "DivideAndConqueror" is to look at the entire protein fold and break up the folding task into manageable build "phases". For example, if you have a topology EEHE, where strands 1-2 are paired, and strands 1-3 are paired, the algorithm might first try to fold E1 and E2 in phase 1, and a structure is produced that passes all filters, the full EEHE is folded in phase 2, with the correctly folded EE remaining fixed. There is an option to control the number of residues that overlap during each build phase.

# Component descriptions

**Architects**
Architects design the fold. Examples of basic architects are the BetaSheetArchitect (designs a beta sheet), HelixArchitect (designs a helix), PoseArchitect (simply uses the input pose), PDBArchitect (pulls a pose out of a PDB file), and BluePrintArchitect (designs based on a blueprint file). Architects can be mixed and matched using the CompoundArchitect. This architect designs a pose using multiple subarchitects (such as those above), each of which designs a part of the overall pose. The CompoundArchitect uses “Connections” to describe how each architect’s design should be joined together.

**Pose Folders**
Pose folders describes how the design created by the architect is turned into a three-dimensional backbone conformation. Only two pose folders currently exists, the RandomTorsionPoseFolder, which inserts random phi/psi values, and the RemodelLoopMoverPoseFolder, which uses remodel to perform 3mer and 9mer fragment insertion.

**PreFoldMovers**
PreFoldMovers are movers run immediately prior to folding by the pose folder. By default, the sidechains of the input pose are saved and the pose is switched to centroid mode. Next, the user-specified PreFoldMovers are run in the user-specified order. Examples of PreFoldMovers are movers that add constraints to the pose or movers that add symmetry. Movers specified in PreFoldMovers must be compatible with centroid mode poses.

**PostFoldMovers**
PostFoldMovers are movers run immediately after folding by the pose folder. The set of movers defined in this section are run first (in user-specified order) on the centroid mode pose after folding. PostFoldMovers should clean up any modifications done by the PreFoldMovers. By default, after the user-specified PostFoldMovers are applied, the pose is switched to full-atom mode, and the sidechains from the PreFoldMovers step are restored. Examples of PostFoldMovers are movers that clear constraints to the pose or movers that remove symmetry and extract the asymmetric subunit. Movers specified in PostFoldMovers must be compatible with centroid mode poses.

**Filters**
In order to consider a folded pose “successful”, it must satisfy the following requirements. 1) The PoseFolder must exit successfully. In the case of the RemodelLoopMoverPoseFolder, if a cutpoint cannot be closed, remodel will fail, and the PoseFolder will also fail. 2) The pose must pass all user-specified filters. Examples of useful filters include HelixPairing filters, StrandPairing filters, HelixStrandStrand triplet filters, Sheet topology filters, secondary structure filter, distance filters, etc. Filters specified here will be run on full-atom poses.

Important note: Filters specified here MUST be compatible with both partial and full-length poses. The BuildDeNovoBackbone mover uses a “divide-and-conquer” strategy that tries to correctly build different pieces of the pose. Therefore, the input to the filter may be a pose that contains 30 residues of a 100-residue design, or all 100. There are no guarantees as to which residues will be folded first. So, care must be taken to ensure that the filters do not use residue numbers, or require a fixed-length pose. Many filters, such as SecondaryStrucutre filter, can read the architect’s design from the pose and use that to filter, which is ideal and should always work. You should never use a SecondaryStructure filter (or other filter) with a fixed-length filter criteria, or no structures will pass filtering.

## XML Syntax

BuildDeNovoBackboneMover is fully configurable and usable via RosettaScripts. IMPORTANT: Proper ordering of subtags of BuildDeNovoBackboneMover is now essential -- I needed to do this to get the code working again after the Rosetta XML code was overhauled during the XML XRW. The following syntax applies:

```
<BuildDeNovoBackboneMover name=(&string)
    dump_pdbs=(&bool, false)
    start_segments=(&string "")
    stop_segments=(&string "") >
    <Architect Tag/>
    <PoseFolder Tag/>
    <Perturber Tag/>
    <PreFoldMovers Tag/>
    <PostFoldMovers Tag/>
    <Filters Tag/>
</BuildDeNovoBackboneMover>
```

**Options**

- **start_segments** - Optional. Comma-separated list of named segments which are required to be present in the first build phase. This can be useful to force the DivideAndConqueor to build any template segments you have and any connecting secondary structure elements. If unspecified, the DivideAndConqueror will choose which segments are build in phase 1.
- **stop_segments** - Optional. Comma-separated list of named segments which are required to be present in the last build phase. If unspecified, the DivideAndConqueror will choose which segments are built in the last phase.
- **dump_pdbs** - If true, pdbs will be dumped during the build process for debugging purposes. If false, only final structures that are properly folded and pass all filters will be outputted.

## Architect Tags

The BuildDeNovoBackboneMover takes a single architect which designs the fold. The architect uses the user inputs and the input pose to come up with a "dynamic blueprint" of what the final folded pose might look like. This "dynamic blueprint" is used by the folding machinery.

### PoseArchitect

The Pose Architect will insert the current Pose as is into the new backbone. The pose is split into named secondary structure elements (e.g. H01, L01, E01, L02, H02, etc).

```
<PoseArchitect name=(&string) add_padding=(&bool, true)
secstruct=(&string, "") />
```

**add_padding** : If true, the residues at the chain termini will be kept. If false, the terminal residues will be removed. If true, these terminal residues become "padding" -- that is, they are overwritten by new residues that are added to the termini, but are used to position and orient the new residues.
**secstruct** : By default, the secondary structure of the added segments will be determined by DSSP. If secstruct is set, the user-provided string will be used to determine the secondary structure of the added segments.

**HelixArchitect**
The Helix architect is used to insert a de novo helix into the pose.

```
<HelixArchitect name=(&string) length=(&string, "") />
```

**length** : Specifies the allowed helix lengths. A colon (:) is used to denote a range of length, and commas (,) are used to add individual lengths. For example, length="11:18" allows helices ranging in length from 11 through 18, and length="11,13,15,18" allows the lengths 11, 13, 15, and 18.

**BlueprintArchitect**
The BlueprintArchitect takes a blueprint file and adds a de novo segment to the pose that is defined by the blueprint.

```
<BlueprintArchitect name=(&string) blueprint=(&string) />
```

**blueprint** : (Required) Blueprint file which contains the information to build this piece of the backbone.

**Extras not reported in manuscript: DeNovoMotifArchitect**
*(note: the RosettaScripts interface for this may not be in the current Rosetta master. If you want to use this but it doesn't work, please contact Tom at tlinsky@gmail.com).*
The DeNovoMotifArchitect takes a motif string and adds a de novo segment (e.g. "10HA-1LB-2LG") to the pose based on that string.

```
<DeNovoMotifArchitect name=(&string) motifs=(&string) />
```

**motifs** : (Required) String of motifs that contain the topological information of the new segment. The format for this is defined by the grammar below:

```
{MOTIFS}      := {MOTIF} | {MOTIFS},{MOTIF}
{MOTIF}       := {MOTIF}-{MOTIF_UNIT} | {MOTIF_UNIT}
{MOTIF_UNIT} := {LENGTH}:{SS}{ABEGO} | {LENGTH}{SS}{ABEGO}
{LENGTH}      := &int
{SS}          := "L" | "E" | "H"
{ABEGO}       := "A" | "B" | "E" | "G" | "O" (or any valid abego)
```

For example, "10HA" is a 10-residue helix with abego A. "10:HA" means the same thing. "2LG-1LB-5HA-2LB" is a segment of secstruct LLLHHHHHLL and abego GGBAAAAABB. "10HA,11HA,12HA" means a helix of either 10, 11 or 12. The length will be chosen when the architect is called, and can be mutated during folding using the DeNovoMotifPerturber.

**BetaSheetArchitect**
The BetaSheetArchitect is used to define de novo beta sheets by combining information from StrandArchitects. Sheets are defined spatially by looking at the face of the sheet (this does NOT

use N-->C ordering). Strands are assumed to be paired to the strands that are defined above/below.
The architect automatically adds the appropriate strand pairings to ensure everything works.
The architect will only attempt to build valid sheets, where the fully-build sheet has no unpaired residues. For example, the XML below can produce only two possible valid sheets (5-5 and 6-6) even though there are four strand length combinations (5-5, 5-6, 6-5, and 6-6). This is because the configurations where length of E01 is 5 and length of E02 is 6 contains an unpaired residue.

```
<BetaSheetArchitect name="sheet"  >
  <StrandArchitect name="E01" length="5:6" orientation="U" />
  <StrandArchitect name="E02" length="5:6" orientation="D" />
</BetaSheetArchitect>
```

The BetaSheetArchitect automatically defines the strand pairings needed to properly form the sheet.
Usage of the BetaSheetArchitect is described below.

```
<BetaSheetArchitect name=(&string) sheet_db=(&string, "")
strand_extensions=(&string, "") >
    <StrandArchitect name=(&string) ... />
    <StrandArchitect name=(&string) ... />
</BetaSheetArchitect>
```

**sheet_db** : A path to a database of native sheets, which can be optionally used. If a sheet_db is specified, the sheet will NOT be build de novo -- it will be extracted from the database and directly inserted without modification into the pose being folded. Default is not set, which means the sheet will be built de novo.
**strand_extensions** : A string to inform the BetaSheetArchitect that certain strands will be extended during the build process. In this case, the machinery that finds appropriate pairings will assume that the strands referred to here are added to the Strands defined in the architect. For example, strand_extensions="E01,1;E02,2" means that one residue will be added to E01 by another architect, and two residues will be added to E02 by another architect. In the example above with 5- and 6-length strands, the only valid sheet configuration would be E01 having length 6 (6 from strand architect + 1 from other architect = 7 residues total to be paired) and E02 having length 5 (5 from strand architect + 2 from other architect = 7 residues total to be paired).
“Another architect" means an architect other than the beta sheet architect itself.  So, for example, let's say you have a 4-residue strand in your pose (and those four residues make up the entire chain 1) and you want to extend that strand be 7 residues long, and make that elongated strand the left-most strand in a 2-strand antiparallel de novo sheet. You could use a PoseArchitect to supply the 4 residues like this:

```
<PoseArchitect name="pose" split_by_chain="1" />
```

Then you *could* create a BetaSheetArchitect that has two de novo strands, like this:

```
<BetaSheetArchitect name="sheet" >
```

```
   <StrandArchitect name="E1" length="3" orientation="U" />
   <StrandArchitect name="E2" length="7" orientation="D" />
</BetaSheetArchitect>
```

The beta sheet architect above would be invalid.  There are no length/register shift combinations that allow a fully-paired beta sheet that contains a strand of length 3 paired to a strand of length 7 -- there will always be four unpaired residues on E1. So the architect will fail to find any valid combinations of lengths/register shifts.
The strand extension changes the way the paired residues are calculated and adds 4 to the number of paired residues. So you can use that strand_extensions options to tell the BetaSheetArchitect that 4 residues from strand "E1" are expected to be unpaired. This implies that you will be doing the pairing using another architect other than the BetaSheetArchitect. I don't think that option does anything except changes the way paired residues on adjacent strands are counted. So if you change what I wrote above to this:

```
<BetaSheetArchitect name="sheet" strand_extensions="E1,4" >
   <StrandArchitect name="E1" length="3" orientation="U" />
   <StrandArchitect name="E2" length="7" orientation="D" />
</BetaSheetArchitect>
```

This will result in a valid sheet because you are telling the BetaSheetArchitect that there are four additional residues on E1 (which in this example will be built by the PoseArchitect). Then you would need to connect "pose.1" to "E1" using a ConnectionArchitect with length 0.

**StrandArchitect**

The StrandArchitect can be used to build a de novo strand. It is not recommended to use the StrandArchitect on its own unless you specify ALL StrandPairings properly in the "Pairings" section.

```
<StrandArchitect name=(&string)
                 length=(&string)
                 orientation=(&string)
                 register_shift=(&string)
                 bulge=(&string) />
```

**length** : Specifies the allowed helix lengths. A colon (:) is used to denote a range of length, and commas (,) are used to add individual lengths. For example, length="5:8" allows strands ranging in length from 5 through 8, and length="5,7,9" allows the lengths 5, 7 and 9.
**orientation** : Specifies the spatial orientation of this strand. Valid orientations are '1' (up; U) and '2' (down; D). Two paired strands are antiparallel if the orientation of one is up and the other is down. Two strands are parallel if they share the same orientation (i.e. both U or both D). Has no effect if the StrandArchitect is defined outside of a BetaSheetArchitect.

**register_shift** : Specifies the register shift of this strand relative to the previous strand. Has no effect if the StrandArchitect is defined outside of a BetaSheetArchitect.
**bulge** : Specifies the strand residue numbers at which a beta bulge is allowed. The same format as the "length" option is used.
Unlike the BetaSheetArchitect, the StrandArchitect does not automatically compute pairings. You need to add them manually like this:

```
<Pairing>
   <StrandPairing segments="E01,E02" orient1="1" orient2="2" shift="0"
/>
</Pairing>
```

**CompoundArchitect**

The compound architect combines the work of multiple architects. It takes 1) a list of architects, each of which designs a part of the structure, 2) a list of connections, which define how the objects designed by the architects are connected in primary sequence space, and 3) a set of pairings, which describe desired spatial interactions among the secondary structure segments.

```
<CompoundArchitect name=(&string)  >
    <Architects>
        <Architect1 ... />
        <Architect2 ... />
        ...
        <ArchitectN ... />
    </Architects>
    <Connections>
        <ConnectionArchitect1  ... />
        <ConnectionArchitect2  ... />
        ...
        <ConnectionArchitectN ... />
    </Connections>
    <Pairing>
        <Pairing1 ... />
        <Pairing2 ... />
        ...
        <PairingN ... />
    </Pairing>
</CompoundArchitect>
```

Each architect tag is described elsewhere in this documentation.

**ConnectionArchitect**

The connection architect joins two segments with a user-specified motif.

```
<Connection name=(&string)
            segment1=(&string, "")
            segment2=(&string, "")
            cutpoint=(&string, "")
            motif=(&string)
            ideal_abego=(&bool, 0) />
```

**segment1** : Name of the segment to be joined. The final pose will contain segments in the following order: segment1, connection, segment2. If unspecified, a random available terminus will be chosen.
**segment2** : Name of the second segment to be joined. The final pose will contain segments in the following order: segment1, connection, segment2. If unspecified, a random available terminus will be chosen.
**motif** : User-specified secondary structure and abego motif to use to join the two segments. The format is "NSA", where N is the length, S is the secondary structure type, and A is the ABEGO code. Segments of mixed type can be specified using dashes (i.e. "1LE-1LA-2LG" for a four-residue loop of ABEGO type EAGG). Multiple motifs can be specified an separated by commas (i.e. "2LG,1LB-2LG" will allow loops GG and BGG).
**cutpoint** : User-specified cutpoint. Only used if both segments being connected are fixed relative to one another. The cutpoint value is the residue number within the connecting motif where the cutpoint will be placed. For example, cutpoint="2:4" means that the cutpoint can be placed after the second, third, or fourth residue in the motif.
**ideal_abego**: If true, the user-specified motifs are used only to determine possible loop lengths, and the loop types allowed are computed using the Koga rules. (default=false)
The following example connects "E1" to "E2" using either "1LE-1LA" or "2LG":

```
<Connection name="E1_E2" segment1="E1" segment2="E2" motif="1LE-
1LA,2LG" />
```

### Pairing Tag

Tags defined in the pairings section allow user-specified secondary structure elements to be marked as paired. Movers and filters can then use this information to obtain information about what the user wants to build. For example, the HelixPairing filter uses the HelixPairing tags defined in the Pairings section to determine which helices in the pose should be paired. The different pairing types are described below:

#### HelixPairing

HelixPairing describes a pairing between two helices.

```
<HelixPairing segments=(&string) parallel=(&bool) />
```

**segments** : *Required.* Comma-separated list of the names of two helices that will be paired. For example, if you have two helix architects named 'H1' and 'H2', the appropriate option would be segments="H1,H2". The segments specified here can be created by any architect (i.e. a HelixArchitect is not required).

**parallel** : *Required.* Tells whether the helices will be paired in an anti-parallel or parallel manner. A value of 1 indicates parallel and 0 indicates antiparallel.

#### StrandPairing

StrandPairing describes a pairing between two strands. This is not necessary if you are using a BetaSheetArchitect, but if you are defining strands outside of the BetaSheetArchitect (e.g. with PoseArchitect), you will need to define the pairings using StrandPairing tags.

```
<StrandPairing segments=(&string) orient1=(&int) orient2=(&int)
shift=(&int) />
```

**segments** : *Required.* Comma-separated list of the names of two strands that will be paired. For example, if you have two strands named 'E1' and 'E2', the appropriate option would be segments="E1,E2". The segments specified here can be created by any architect (i.e. a StrandArchitect is not required).
**orient1** : *Required.* Spatial orientation of the first segment listed in the segments option. Valid values are 1 ("UP") or 2 ("DOWN").
**orient2** : *Required.* Spatial orientation of the second segment listed in the segments option. Valid values are 1 ("UP") or 2 ("DOWN").
**shift** : *Required.* Register shift between the first and second strands defined in the segments option. All positive and negative integers are acceptable. A register shift of 0 means that the residue furthest 'DOWN' on strand 1 is paired with the furthest 'DOWN' residue on strand 2. A register shift of -1 indicates that the residue furthest 'DOWN' on strand 1 is paired with the second-furthest 'DOWN' residue on strand 2. A register shift of 1 indicates that the residue furthest 'DOWN' on strand 1 is not paired to strand 2, but the second-furthest 'DOWN' residue on strand 1 is paired with the most 'DOWN' residue on strand2.

## Perturber Tag

*Note and warning: It is likely that tag-parsing for perturbers is still broken after the Rosetta XML overhaul in 2017. I tried to fix this shortly after the rewrite, but it was difficult to port to the new system.*

The perturbers allow the design created by an architect to change during the build process. For example, a HelixPerturber will change the length of a helix to any user-allowed value prior to folding the pose. Despite the plural name, only one perturber is allowed. The effect of this would be a different helix length being sampled for every fold attempt. If a perturber is not used, the BuildDeNovoBackboneMover will randomly choose a helix length when it is called, and will not change the value until it is called again. One highly useful case for perturbers is to perturb loop connections. This allows a different loop type to be sampled every time a fold attempt occurs.

```
<CompoundPerturber>
  <Perturber1 />
  <Perturber2 />
  ...
  <PerturberN />
</CompoundPerturber>
```

**Example**

```
<CompoundPerturber>
  <HelixPerturber architect="H1" />
  <ConnectionPerturber architect="H1_E1" />
  <ConnectionPerturber architect="E1_E2" />
  <ConnectionPerturber architect="E2_E3" />
  <ConnectionPerturber architect="E3_E4" />
</CompoundPerturber>
```

**HelixPerturber**

The helix perturber alters the length of a given helix to a valid user-specified value.

```
<HelixPerturber architect=(&string) />
```

**architect** : Name of the architect to perturb. Must be a HelixArchitect.

**DeNovoMotifPerturber**

The DeNovoMotif perturber alters the motif selected by a DeNovoMotifArchitect to a motif randomly selected from a DeNovoArchitect’s list of user-specified motifs.

```
<DeNovoMotifPerturber architect=(&string) />
```

**architect** : Name of the architect to perturb. Must be a DeNovoMotifArchitect.

**ConnectionPerturber**

The connection perturber perturbs a connection to a valid user-specified value. If ideal_abego=1, a loop that is 'ideal' according to the Koga rules will be inserted, with length in the user-specified range of loop motif lengths.

```
<ConnectionPerturber architect=(&string) />
```

**architect** : Name of the architect to perturb. Must be a ConnectionArchitect.

## PreFold Movers Tag

The PreFold movers tag allows the user to call arbitrary movers after the pose is converted to centroid mode, but before the folding attempt occurs.

```
<PreFoldMovers>
    <Add mover=(&string) />
    ...
    <Add mover=(&string) />
</PreFoldMovers>
```

## PoseFolder Tag

The PoseFolder defines the method by which the backbone torsions will be generated. The different folding methods are described below:

### RemodelLoopMoverPoseFolder

The RemodelLoopMoverPoseFolder uses the remodel loop mover to fold the structure via fragment insertion.

```
<RemodelLoopMoverPoseFolder scorefxn=(&string) />
```

**scorefxn** : ScoreFunction to use during folding. Must be compatible with centroid poses.

### RandomTorsionPoseFolder

The RandomTorsionPoseFolder is mainly intended for debugging purposes. It inserts random values for the backbone phi/psi angles.

```
<RandomTorsionPoseFolder />
```

## PostFold Movers Tag

The PostFold movers tag allows the user to call arbitrary movers after the pose is folded and still in centroid mode, but before the folding attempt occurs.

```
<PostFoldMovers>
    <Add mover=(&string) />
    ...
    <Add mover=(&string) />
</PreFoldMovers>
```

## Filters Tag

The Filters tag allows the user to call arbitrary filters to assess the quality of the folded pose. The filters should be compatible with partially-built poses. Examples of filters that are useful are: GeometryFilter, SecondaryStructureFilter, HelixPairingFilter, HelixKinkFilter, SheetTopologyFilter, and presumbably others.

```
<Filters>
    <Add filter=(&string) />
    ...
    <Add filter=(&string) />
</Filters>
```

## Post-build Analysis

We created a script (see below) saves a csv file that includes secstruct, abego, and length of segments built by specific architects. This is useful for determining which lengths/secstructs/abegos worked best to form “folded” structures.

```
#!/usr/bin/python
import sys
import argparse
import pandas as pd
from xml.dom import minidom

## @brief StructureData parser
class StructureData( object ):
      ## @brief Create StructureData object from input
      #  @param[in] pdb           PDB file containing data
      #  @param[in] remarks       PDB remark lines containing data
optional
      #  @details Either 1) pdb, or 2) remarks, must be specified
      def __init__( self,
            pdb=None,
            remarks=None,
            ):

      if pdb and remarks:
            raise RuntimeError( 'StructureData(): You cannot specify
both pdb and remarks' )

      if pdb:
            assert not remarks
            self.from_pdb( pdb )
      elif remarks:
            assert not pdb
            self.from_remarks( remarks )
      else:
            raise RuntimeError( 'StructureData(): You must specify pdb
or remarks' )

      def __repr__( self ):
      return self.to_xml()

      def from_pdb( self, pdbfile ):
      with open(pdbfile) as f:
            lines = [x.strip() for x in f if x[:10] == 'REMARK 994']
      self.from_remarks(lines)
```

```
    def from_remarks( self, remarks ):
    xml = ""
    for remark in remarks:
        fields = remark.strip().split()
        if len(fields) < 2:
            continue
        if fields[1] != "994":
            continue
        newline = remark[11:]
        if newline[-1] != '#':
            xml += newline + "\n"
        else:
            xml += newline[:-1]
    self.from_xml( xml )

    def to_xml( self ):
    return self.dom.toxml()

    def from_xml( self, xml ):
    self.dom = minidom.parseString( xml )

    def resid( self, segment, residue ):
    elements = [ x for x in self.dom.getElementsByTagName(
'ResidueRange' ) if x.attributes['name'].value == segment ]
    assert len(elements) == 1
    start = int(elements[0].attributes['start'].value)
    return start + int(residue) - 1

    def alias( self, alias_name ):
    elements = [ x for x in self.dom.getElementsByTagName( 'Alias' )
if x.attributes['name'].value == alias_name ]
    assert len(elements) == 1
    return self.resid( elements[0].attributes['segment'].value,
int(elements[0].attributes['res'].value) )

    def full_secstruct( self ):
    elements = [ x.attributes['ss'].value for x in
self.dom.getElementsByTagName( 'ResidueRange' ) ]
    return ''.join( elements )

    def full_abego( self ):
    elements = [ x.attributes['abego'].value for x in
self.dom.getElementsByTagName( 'ResidueRange' ) ]
    return ''.join( elements )
    def secstruct( self, segment ):
    elements = [ x for x in self.dom.getElementsByTagName(
'ResidueRange' ) if x.attributes['name'].value == segment ]
    assert len(elements) == 1
    return elements[0].attributes['ss'].value

    def abego( self, segment ):
    elements = [ x for x in self.dom.getElementsByTagName(
'ResidueRange' ) if x.attributes['name'].value == segment ]
    assert len(elements) == 1
    return elements[0].attributes['abego'].value


def options(argv):
    parser = argparse.ArgumentParser(description="Analyzes structures
created with the BuildDeNovoBackboneMover")
    parser.add_argument('--segments', required=True, type=str,
nargs='+', help="Segment names to analyze")
    parser.add_argument('--pdbs', required=True, type=str, nargs='+',
help="Built PDBs to scan")
    parser.add_argument('--output', type=str,
default='structure_data.csv', help="Output csv file")
    return parser.parse_args(argv)

def main():
    opts = options(sys.argv[1:])

    info = []
    for pdbfile in opts.pdbs:
    sd = StructureData(pdb=pdbfile)
    data = {}
    data['pdb'] = pdbfile
    for segment in opts.segments:
        data[segment + '_ss'] = sd.secstruct(segment)
        data[segment + '_abego'] = sd.abego(segment)
        data[segment + '_len'] = len(sd.secstruct(segment))
    info.append(data)

    df = pd.DataFrame()
    for col in sorted(info[0].keys()):
    df[col] = [x[col] for x in info]
    df.to_csv(opts.output, index=False)
    print("Wrote {}".format(opts.output))
```

```
if __name__ == '__main__':
      main()
```

## Symmetry

The FoldArchitect is compatible symmetry operations, one needs to generate a symmetry defining file as with all Rosetta symmetry.