A tutorial/documentation guide on using Dimeric MultiDomain Biosensor Builder

Lucas Rudden, December 2023

The following guide will take you through an example of building a dimeric biosensor receptor chimera of SCFR with TpoR using our assembly guide protocol. Be prepared for this to use a fair amount of computational resources and take some time, as sampling the vast conformational space these large multi-domain receptors can occupy during assembly is a challenge. If you're running this just as a proof of concept, then please go ahead and reduce the number of samples generated at each stage. The scripts discussed here all related to those on our GitHub:

`https://github.com/barth-lab/Dimeric_MultiDomain_Biosensor_Builder`

If you have any questions or concerns, then please either flag a ticket on the GitHub, or contact me directly: `lucas.rudden@epfl.ch`

1 PREREQUISITES

This package must be run on a GNU/Linux or other unix-based operating system. You must have pre-installed Rosetta to run the actual domain assembly sections.

`https://new.rosettacommons.org/demos/latest/tutorials/install_build/install_build`

The bash scripts used as part of the assembly sequence (outlined below) rely on Python scripts. These feature full comments/argument instructions within, so you are more than welcome to use the Python scripts if you wish to avoid the bash command line approach. You must have `Python 3.x` installed (*i.e.* not `Python <=2.7`).

The following Python modules must be installed. All are installable via conda, and we recommend using a conda environment anyway to keep your system safe.

- numpy
- biobox (https://github.com/Degiacomi-Lab/biobox)
- argparse
- pandas

2 Initial Stages

In order to begin the assembly protocol of multi-domain membrane receptors, you must first begin with some prerequisite structures (i.e. domains) that you wish to assemble in a desired order. This ideally includes also the TM dimerising domain. Note that this protocol is to be applied to dimeric systems, the original domain assembly protocol in Rosetta should work for monomers (although by all means, you can apply this protocol to monomers), and multimers require a more careful approach that this method currently doesn't trivially offer. For now, you must also have the full sequence in mind of your resultant chimera, including the linker connective regions.

To begin assembly (as a bare minimum), you must have the following pre-prepared:

1. The PDB structures of your domains (idealised/relaxed through Rosetta)
2. A file containing connective linker sequence information (e.g. linkers.txt)
3. A file containing resid positions between which you want residues removed (e.g. remove_linkers.txt)
4. (in theory) Fragment files

To describe what these files should look like, we will take the example of an SCFR-TpoR chimera (Figure 1) as a model receptor similar to that you might be working with. You can find the necessary structural files in the example/ folder of the repository. There is an absence of the CT domains in this structure as we have no structural data for them, and they are unnecessary anyway as you can calculate the coupling strength between the LBD and TM domains as a proxy for communication through the receptor.

The dimerising domains must be **pre-dimerised** prior to input for the algorithm to work. This includes the ligand binding domain (LBD), where the two sides of the binding domain of the receptor may be connected only by the soluble ligand.

Let's discuss each of the needed files in turn.

3 Files

3.1 Domain PDBs

These are your input PDBs for each of the domains being assembled (so in the above example there would be 7). These could be sourced from the PDB, an AlphaFold structure, homology model, etc.. Each should ideally be idealised and relaxed in Rosetta or at least prepared for Rosetta input.

Nothing is stopping you from preemptively combining domains into a larger structure to ease the computational calculations if you know how they should organise themselves. This is particularly pertinent if you have two monomeric units adjacent to one another that are from the same receptor.

We are going to do just that. Since we know D1, D2 and D3 will have limited mobility with respect to one another while bound to the ligand, we're going to combine them into one large

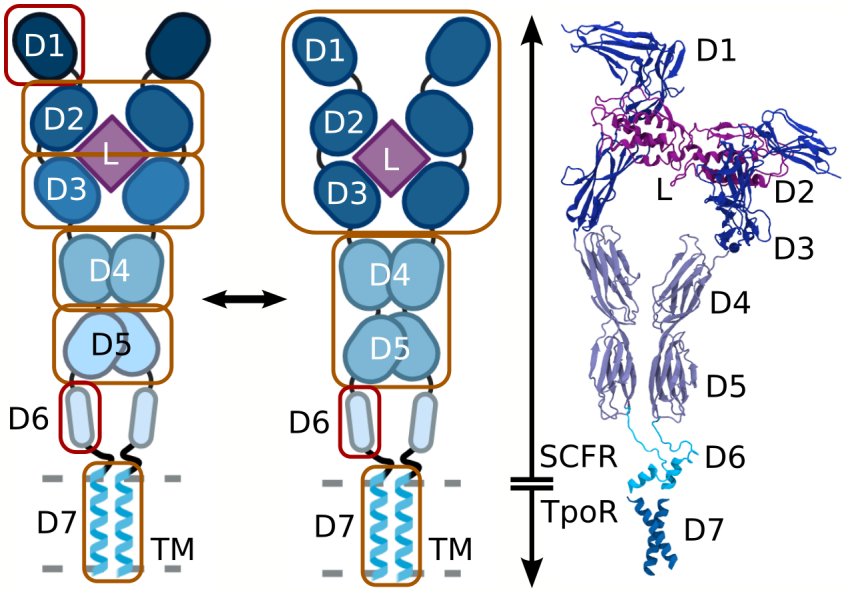

FIGURE 1
An example schematic for the protein we will scaffold. D denotes domain, moving from domain 1 to domain 7, with D2/3 the ligand binding domains (LBD) and D7 the TM. An orange outline box indicates a dimerising input, which includes the LBD, while a monomer domain is highlighted via the red box. The ligand is coloured palatinate, while the respective receptor domains shades of blue. When simplifying the design problem, we can group up several domains into one, hence the larger boxes on the right. On the far right, we demonstrate a rendering of the chimera we will be scaffolding, coloured by the grouped domain.

PDB. We will treat D4 and D5 similarly since both are restricted in their conformational space with respect to one another owing to the beta-sheet link that connects them, and the intermolecular interface on both. We therefore end up with four PDBs overall, D13, D45, D6 and D7 (the example PDBs given to you for this tutorial correspond to these four).

Of course, if you are creating a novel chimera, then be careful with this "hack". If you have any concerns about conformational diversity between two domains, for example if you have a long linker connecting them, then avoid grouping structural domains.

3.2 LINKER POSITION REMOVAL TXT FILE

The linkers in Figure 1, shown as the black connecting lines between domains, will all need to be rebuilt by Rosetta. As input, you can either pre-emptively remove these linker regions, or you can ask the first stage of the protocol to do this for you. For this, you still need to know which linkers are to be removed so it is arguably easier to remove them in pymol or VMD beforehand. An example for input in the remove_linkers.txt file for the above scaffold could be:

```
1 3 X X 200 202 X X
1 4 59 60 61 64 119 120
X X 94 96
X X X X X X X X
```

Notice the number of lines (4) corresponds to the number of domains (where we've lumped

D1/2/3, and D4/5 together in our inputs). Each line chronologically relates to a domain moving from the outer extracellular region to the intercellular region. Dimers have 8 corresponding values, while monomers have 4. If we take the first line of the above example and read character by character:

```
1 3 X X 200 202 X X
```

The first two characters, `1` and `3`, specify that we want to remove residues 1-3 from the start of the first domain (i.e. the top of the PDB). The following two `X`s mean no residues are to be removed from the end of the domain. But because there are actually two receptor domains in this PDB, we need to do the same for the second domain. Note we are assuming here the ligand is at the end (e.g. labelled as chain C), of course it could be in any arbitrary order, as long as your given residue numbers corresponds to those in the actual provided PDB file. So for the second domain, we want to remove the same first two initial residues - hence cut between `200` and `202`, while preserving the end of the domain.

The same logic is extended for the other lines, including the monomer, except now you don't need a second set of 4 digits since there is only one composite domain. If you pre-remove these linkers anyway, then simply label each character as an X - as we have done in the tutorial example, but note that this file is always required.

3.3 CONNECTIVE LINKER TXT FILE

This file tells the auto builder what the sequence (if any) of connective linkers should be before and after the start of domains. This is necessary for both the domain assembly protocol to fill in the linkers and for the end linker rebuilding stage. Taking the above example in Figure 1:

```
1 X DKG
2 X N
3 X X
4 RVETATETAW X
```

Again we have 4 lines for each domain. The character in the first column denotes the domain number (in order of input). Each line features only two additional key pieces of information since we're just telling the auto builder the sequence to be added, and therefore don't need to provide redundant information for each sub-domain in a dimeric domain, unlike in the remove_linkers.txt case where the specific residue numbers were different.

Taking the first line as an example again, the `X` after the domain number denotes that no linker is to be added at the N termini of the PDB. This makes sense, this is the first domain after all so we're not trying to connect anything here anyway. The second column, `DKG`, says that at the end of **both** sub-domains in this dimer, we want to add the linker DKG. This linker will connect to the start of domain 2, hence the start of the linker on domain 2 (the second line) features just an `X`. We could split this linker in any way between the end of domain 1 and start of domain 2, and the auto builder will auto-concatenate the two sequences, but for readability, I normally leave one side to be an `X`. This logic applies in reverse, for example between domain 3 and 4 the connective linker information is provided at the start of domain 4 and nothing is given for the end of domain 3.

When reading this file and the above PDBs, the autobuilder will decide how long constraints need

to be to support the scaffold and prepare the necessary fasta inputs. In other words, the constraint and fasta files needed by the domain assembly protocol are generated here automatically.

3.4 FRAGMENT FILES

These files are **not** needed to prepare any inputs, but are needed for the Rosetta domain assembly side of things so it can construct the various possible linkers. Indeed, I would recommend waiting until after you've run the first stage of preparation before generating any fragment files so you have the correct fragment files.

These files can be generated independently, e.g. with Robetta at:

`http://old.robetta.org/fragmentqueue.jsp`

Or using the FragmentPicker mover within Rosetta.

4 RUNNING THROUGH AN EXAMPLE

Let's run through the above example. We can't construct the whole scaffold in one shot, we need to do things in parts to make the problem manageable.

4.1 INITIAL SCAFFOLD

You can see in Figure 2 that the first stage of assembly needs to build a continuous "thread" of domains, which involves hopping to the other side of the receptor at each dimerisation domain (look at where the linkers are going to be built between D3 and D4, or D5 and D6 during the assembly process). Therefore, it ends up missing out on the second D6; we will need to add this in later. To take into account this missing monomeric domain and the missing linkers during assembly, constraints are included between the appropriate domains that roughly accommodate the missing domain size and linker lengths, with some tolerance - the first between D1/2/3 and D4/5, the second between D5 and D6. The linker lengths are calculated based on sequence length[Choi, Agarwal and Deane 2013].

Now we will run through the first command of the assembly protocol. Say we are in the folder containing the necessary PDBs, called D13, D45, D6 and D7.pdb, respectively, as well as linker position removal file (Section 3.2, remove_linkers.txt), and our connective linker file (Section 3.3, add_linkers.txt):

```
/location/to/protocol/main/prepare_scaffold.sh \
                      -T 4  \
                      -s "D13.pdb D45.pdb D6.pdb D7.pdb" \
                      -l remove_linkers.txt \
                      -d "1 2 4" \
                      -a "1 2 3 4" \
                      -x add_linkers.txt \
                      -L True
```

The \ at the end of each line except the last just tells bash not to run the command yet (*i.e.* there are more arguments we wish to specify).

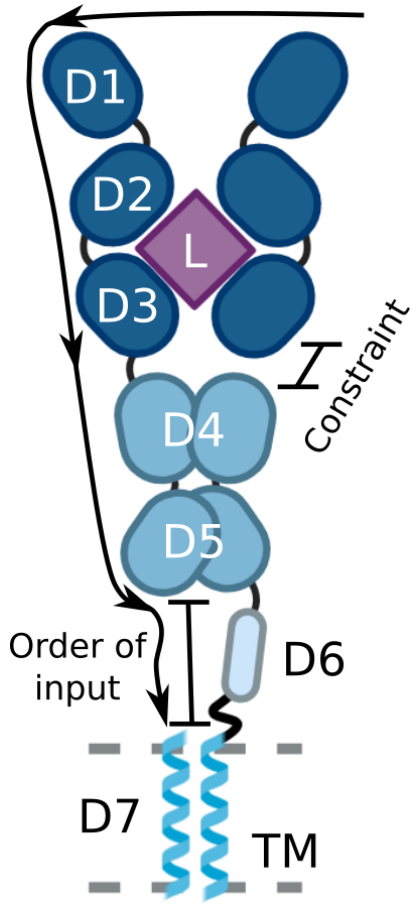

FIGURE 2
The first scaffold we will construct with the domain assembly protocol. The arrow denotes how the domain assembly protocol will "read" our inputs. Each dimerisation domain is essentially a crossing point where we hop to the other side of the receptor. There are two constraints that will be defined, one between D1/2 and D3 (based on a connecting linker), and one between D3 and D5 (based on the missing domain and two connecting linkers).

The first argument should be the location of the transmembrane domain in terms of your input PDBs, so while it's the seventh domain we're inputting because the first three domains and fourth/fifth domains are grouped together, we set `-T` to 4.

`-s` refers to our input domain file names. This should always be contained in a string, *i.e.* within speech marks. We list the input domain PDBs in order of our desired assembly shown in Figure 2. `-l` is simply the remove_linkers.txt file we discuss above. `-d` lists our dimerisation domains with respect to order of input. It is crucial this is correct and in order. In our assembly, and in our order of inputs given by `-s`, domains 1 2 and 4 are the dimerisation domains.

There are two strategies for dealing with the linkers on our PDBs. You either pre-emptively remove the linkers from the PDBs or you use only the remove_linkers.txt file to instruct what linker positions need to be removed. In both cases, you will need add_linkers.txt to tell the autobuilder which linkers to rebuild and where. In this tutorial, we will be applying the first strategy. If you want to completely discard those linkers and not rebuild them back in, `-a` provides this option. Any domain **not** specified by the `-a` flag will effectively have their linker

sequence discarded according to remove_linkers.txt. Since we want to rebuild the linkers between all domains, we have specified every domain under the `-a` flag as a string. Unless you are very careful, it is recommended that you **always** specify all domains with this flag (like in the example).

`-x` gives the filename for add_linkers.txt, which contains the information on the sequence identity of our rebuilt linkers, as discussed above.

Finally, `-L` tells the assembly protocol whether a ligand is present in your PDB structure - for now this needs to correspond to the first domain structure you're inputting (D13 in this example), the ligand itself must come first in the metadata of the PDB file. If you have no ligand in your structure, then you can remove this flag.

Running this command should take about a minute to run. After that, you should find a new folder called run`TIME_STAMP` where `TIME_STAMP` is the `UNIX` time from when you ran the command. Inside this folder, you'll find your initial input files copied and a folder called input_scaffold. It is in this that your processed files are located, and the one the actual first stage of assembly will run from.

input_scaffold should therefore contain your PDB files (with the _cut suffix to indicate there may have been some minor changes from your inputs), two fasta files and the constraint file (cst). The first fasta file, all.fasta, contains the fasta of the structure that will be constructed at the end of the first assembly cycle, *i.e.* it corresponds directly to Figure 2. The second fasta file, all_verbose.fasta, is for your benefit and for the following stages of the autobuilder. It contains the name of each domain followed by its sequence and keeps the linker information between domains on separate lines.

Before we run the actual first stage of domain_assembly in Rosetta, you will need to move the fragment files into this input_scaffold. It is the all.fasta file which can help you generate the necessary files in this instance using the website provided in Section 3.4.

The first stage of the assembly requires the mp_domain_assembly RosettaMP application. The provided script, mp_assembly_stage1.slurm in HPC_main, is set up at the moment to be run on a High-Performance Cluster (HPC), specifically in a slurm environment, to expedite the calculation since dozens of runs will be required simultaneously. You can find more information on this application here:

```
https://www.rosettacommons.org/docs/latest/
        RosettaMP-DomainAssembly
```

I won't cover every Rosetta-specific flag as you can find more information on these on the website or the Rosetta Slack group, but I will describe some of the flags in the script you will likely need to adjust.

Take a look at mp_assembly_stage1.slurm script inside the HPC_main folder. I would recommend copying this script into the base run folder. Notice we're going to run 100 instances of the domain_assembly on a serial queue (given by the SBATCH commands at the top of the script). In other words, nothing will be run in parallel for now. Three variables must be pre-defined, the TM domain location (4 in our example), the location of Rosetta, and the domain input PDB names. So, for example:

```
TM=4
R=/data/scratch/lrudden/Rosetta/
domains="input_scaffold/D13_cut.pdb input_scaffold/D45_cut.pdb
         input_scaffold/D6_cut.pdb input_scaffold/D7_cut.pdb"
```

Make sure you're in the base run folder for the paths to be correct. Other than that, we just make sure that the for loop is correct with respect to the size of the array specified at the top (*i.e.* 100). Then we can run on the HPC with:

```
sbatch mp_assembly_stage1.slurm
```

The run time varies depending on the size of the system. For a large assembly like that described above, we can expect the total runtime to take 3 days easily.

4.2 CLUSTERING

The output of the first stage of assembly should generate 100 silent files (if you set the size of the array to 100), each containing 100 structures. Those 10000 structures now need to be clustered.

Copy over the output_scaffold folder somewhere safe to work on. We are going to do a little cleaning before clustering; specifically, we're going to remove any structures that violate the constraints we defined during stage 1. Yes, some structures will violate your constraints because they are just an extra energy penalty in the Rosetta scoring function. This can be tricky later on, as if you have very tight constraints the final linker rebuilding stage can be problematic (more on this later). In these circumstances, it is often safer to increase the thresholds within the constraint file and rerun mp_assembly_stage1.slurm

Given how many structures we have, there are two filtering steps prior to clustering. The first is the constraint violation check; the second is to take the top 10% of structures as determined by energy. This occurs automatically, so if you have a lot of issues with constraint violations leading to few final structures, it might be useful to ignore the 10% slice.

```
/location/to/protocol/main/remove_constraint_violations.sh
```

Provided you're in the folder with the silent files, this should work first time. If the output silent file (filtered.silent) is empty (which should be rare!), that means no structures you generated satisfied your constraints. You should increase the constraint threshold within the cst file by adding $\sim$5 Å and rerun the assembly if this is the case.

Now let's cluster. Make sure to run this from the output_scaffold folder:

```
/location/to/protocol/main/run_clustering.sh \
                     -R /location/to/Rosetta \
                     -S filtered.silent
```

Where `-R` is your location to Rosetta, while `-S` is the input silent file name. You may need to edit the Rosetta flags/linuxgcc installation type in run_clustering.sh depending on your exact Rosetta configuration.

This will run the base Rosetta clustering algorithm. More information on the Rosetta clustering application is given at:

https://new.rosettacommons.org/docs/latest/application_documentation/utilities/cluster

Clustering is done based on RMSD between structures assigned to clusters, not energy. These clusters have radii in Å that can be defined by user input, but in the absence of a given value Rosetta automatically determines it. This radius is normally too large. Therefore, I recommend rerunning the clustering with a set of smaller radii between 50% and 80% of the value determined by Rosetta. If you set the flag `-p` to anything other than 0 (which is the default) in the above run_clustering.sh script, it will return to you a set of suggested radii to rerun the clustering algorithm with. You can then include that rerunning the clustering script with:

```
/location/to/protocol/main/run_clustering.sh \
                      -R /location/to/Rosetta \
                      -S filtered.silent \
                      -C 9.5
```

There is also a script in lib that can re-report possible clustering radii. Say your provided radius was 9.5 Å, simply run:

```
/location/to/protocol/lib/calc_radius.sh 9.5
```

This should return provide you with a set of radii that can be used to repeat the clustering several times. For this though, it may be pertinent to run on the HPC once more so you can make the most of its parallelisation. There is a slurm batch script to help with this also in HPC_main/mp_cluster.slurm which takes as its input the silent file and radius size. You will need to edit the Rosetta location inside the script before using it.

When running this slurm script, you must be in a new folder, based in the output_scaffold folder, with the name of the radius value itself, e.g. /output_scaffold/5.4/. So, provided you are in the folder /output_scaffold/5.4/, you can run:

```
sbatch /location/to/protocol/HPC_main/mp_cluster.slurm \
       ../filtered.silent 5.4
```

Alternatively, given your initial Rosetta predicted radius (9.5 Å in this example), you can automatically generate possible radii, create their respective folders and run the slurm script in bulk. If, instead, you are now in the output_scaffold folder on your HPC:

```
/location/to/protocol/HPC_main/loop_cluster.sh \
        filtered.silent 9.5
```

After the run has finished, which should take roughly half a day, you will find each folder named by their respective cluster radius will contain a series of PDBs names c.*n*.*x*.pdb. *n* is the cluster number the PDB is assigned to, and *x* is index of that PDB within that cluster. Note there is nothing special about the c.*n*.0.pdb structures, they define the cluster centres but that is because they occurred first in the silent file.

Copy these cluster folders back into the output_scaffold folder on your work PC. As a ballpark heuristic from this output, you're looking for at least 20 clusters where between 10-20% of all structures are in the largest cluster and at least 1-5% in the smallest. This is by no means a fixed rule. There is a script in lib to help you check this information that you can run from the output_scaffold/ folder (now with the clustered data in various radii folders).

```
/location/to/protocol/lib/sort\_clusters.sh 9.5
```

Where again, you need to specify your initial radius. From the output choose your best cluster radius to represent the data. These cluster centres will be your inputs for the next stage, albeit the files are taken automatically (see below). This is not an ideal clustering approach, and we're experimenting with other methods to cluster the data better. But for now, it should provide a decent spread of models for the next stage of construction.

4.3 FILLING THE MISSING DOMAINS

The output scaffolds are now an input "domain" in the next round of assembly. That means any additional domains can't be arbitrarily inserted into the scaffold. Luckily, we can trick the assembly into thinking an additional domain is either at the N or C termini. Unfortunately, it means if you have multiple missing domains in multiple locations, you will have to repeat this step to fill in the missing domains.

In our example, we need to add in D4. If there was a monomeric domain above D4 (e.g. D3 was monomeric), you could, in principle, add both D3 and D4 at this stage. But if D3 was dimeric, and D2 was monomeric, you would need to separate D4 and D2's inclusion into two separate rounds of assembly.

D4 will be connected to the TM domain through a rebuilt linker before an imaginary link is created between the end of D5 and the start of D1 (see Figure 3). This entails some internal reshuffling of our clustered PDB topology, as we need to extract the relevant half of the TM domain and move it before the D1-4 domain in our c.*n*.*x*.pdb set, but thankfully we have an automated means to do this.

Because D4 will be inserted at the N termini, there will be a missing linker above it that would nominally connect it to D3 (see Figure 3). Our protocol will automatically generate the needed constraint file.

From the base run folder that contains the input_scaffold and output_scaffold, we can run the next stage of the protocol:

```
/location/to/protocol/main/assemble_domains.sh \
                      -d "1 2 4" \
                      -s "D4.pdb c.0.0.pdb" \
                      -C 5.4 \
                      -p 3
```

Let's go through the flags individually. The first few are familiar: `-d` provides a string of the dimerisation domains in the scaffold topology, given that D1 and D2 are grouped up as single dimerising domain (see prepare_scaffold.sh above). `-s` are the PDB files we want to assemble. So, D4.pdb is our missing domain, while c.0.0.pdb is the generic name of the first cluster centre. Given a provided cluster radius size (provided by `-C`), the protocol will automatically grab all cluster centres, i.e. those given by c.*n*.0.pdb, for use in the next stage. Finally, `-p` is the position in the current order of domains we need to insert our domain. Here, it is given as position 3 since D1 and D2 are grouped together. There is no need to specify linkers or any other information here; all metadata required is extracted from the pre-existing all_verbose.fasta file.

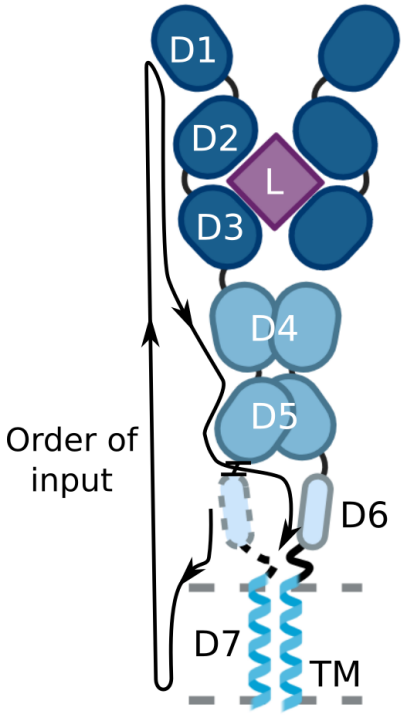

FIGURE 3
The second scaffold we will construct. Specifically filling in domain 4 (shown via the dotted line). Note the order of input has changed from before, now we specify D4-D5-D1 (dimer)-D2-D3 (dimer)-D4-D5. There is a constraint included between D3 and the inserted D4 domain.

This command should produce a new folder called input_scaffold_2, in which you will find similar files to input_scaffold: all.fasta with the total fasta of the topologically shuffled desired output structure, all_verbose.fasta as a human-readable format of all.fasta, the various PDB input files, including D4.pdb but also all cluster centre files, the generated constraint file called cst, and the frag files copied from input_scaffold. This folder will need to be copied to your HPC for the next stage of assembly. Make sure it is placed in the same base folder as input_scaffold and output_scaffold.

Once you have copied everything over, it is time to submit the next assembly slurm script. We're going to use the mp_assembly_stage2.slurm script inside the hpc_run folder of main. Currently, the script is setup to run 10 repeats per input structure (the i loop in the script). You may want to adjust this depending on your computational hardware, as the multiple of this with the number of cluster centres - 30 in the script example, gives you the total number of instances you're going to run on the HPC. In other words, this script will submit 300 jobs. Depending on how many cluster centres you have, check input_scaffold_2 to check this, you will need to change the j loop maximum number. Since we're looping from 0, you need to set it to the maximum number of n in c.n.0.pdb, minus 1. The only remaining changes are: The assignment of the TM domain in the input (2 in this example since we list the TM containing c.n.0.pdb second in `-mp:assembly:poses`), and the additional domain we're including:

```
domain0="input_scaffold_2/D4_cut.pdb"
```

After this you're free to submit the job as before from the base run folder:

```
sbatch mp\_assembly\_stage1.slurm
```

Again, the runtime will likely be around 3 days.

4.4 FINALISING THE STRUCTURE

In our example we have completed the formal assembly part of the protocol. However, it may well be that you have additional missing domains in your structure depending on the topology. If this is the case, you will need to repeat the above clustering / mp_assembly_stage2.slurm stages until all domains are present.

While we have now in principle completed the assembly, we still need to fill in the final missing linkers (Figure 4) and ultimately relax our structures before performing any final assessment. Prior to building in the missing linkers, you will need to cluster your output from the previous assembly protocol. You can follow the same approach as that given in Section 4.2.

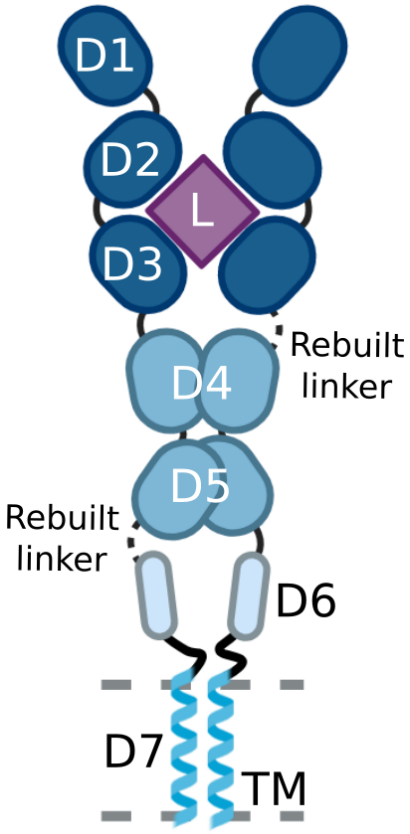

FIGURE 4
Final assembly with the missing linkers still to be rebuilt demonstrated via the dotted lines.

Once you have decided which cluster centre, and by extension which cluster representative structures, to use moving forward, we can run the build_linkers.sh script located within the main folder. This will prepare the metadata for Rosetta to use when building the missing linkers. From the base run folder we've been running everything from by now, you can run the script with the following:

```
/location/to/protocol/main/build_linkers.sh \
```

```
-R /location/to/Rosetta \
-C 5.4 \
-d "1 2 4" \
-o "" \
-l 1 \
-r _2
```

Some of these flags are as before. `-d` are the dimerisation domains in the original input topology, while `-C` defines the cluster centre you're taking forward.

After copying the produced input_loop/ and output_loop/ folders to your HPC, you can then run the final macro-assembly build command from the root run/ folder.

```
sbatch /location/to/protocol/HPC_main/build_buildloops.slurm
```

Feel free to change the number of macro-loops (given by the i parameter in the script) to modulate the number of repeats per system.

When this is complete, you have your final set of receptors.

5 POSSIBLE ERRORS

The following list of common errors and issues I often encountered while running the protocol in bulk. Hopefully, they'll help you address any problems in your set up so you're not banging your head against a wall.

1. During the prepare_scaffold stage, if you have an extra space at the end of any line in a text file (e.g. remove_linkers.txt), this can cause a character read error based on the "\n" that will be read. Just make sure your last character is an actual needed character - e.g. an X, or DKG.

2. You might receive a Rosetta error during assembly with wording akin to:

   ```
   /location/to/rosetta/source/bin/mp_domain_assembly.
   linuxgccrelease: error while loading shared libraries:
   libzlib.so: cannot open shared object file:
   No such file or directory
   ```

   This means there are critical libraries Rosetta is unable to access. You can include those libraries with something like the following appended towards the start of your slurm script:

   ```
   export LD_LIBRARY_PATH="path/to/rosetta/source/build/
   external/debug/linux/3.10/64/x86/gcc/4.8/
   default/:$LD_LIBRARY_PATH"
   ```

   This will be different depending on your compilers etc. that you used for Rosetta. Crucially, these libraries will always be located somewhere in that /source/build folder. You can find exactly where these libraries are that need to be included with a find:

   ```
   find -name "*libzlib.so"
   ```

REFERENCES

Choi, Yoonjoo, Sumeet Agarwal and Charlotte Deane (Feb. 2013). 'How long is a piece of loop?' In: *PeerJ* 1, e1. DOI: 10.7717/peerj.1.