

A Introduction

SoS Explorer allows an architect to computationally produce optimal architectures for systems of systems including cyber-physical systems. The solutions (architectures) are represented as graphs where the systems are nodes and the interfaces between systems are edges. In order to produce optimal architectures, a set of objectives is given to an optimization method. The set of objectives embodies the key performance measures (KPMs) of the architecture. In order to calculate the objectives, information regarding the systems is required. In the approach taken by SoS Explorer there are three categories of information associated with each system:

- Characteristics—Quantifiable attributes such as cost or performance parameters. These are real (floating point) values.
- Capabilities—Functionality required by the system of systems. These are Boolean values representing which ones are present in the current systems.
- Feasible interfaces—Stipulates which systems may interface with one another. These are Boolean values.

Figure 1 shows SoS Explorer after it is opened. It is comprised of three columns. The left column allows the entry of the systems and their associated characteristics, capabilities, and feasible interfaces. It also allows a description of the problem to be included. The middle column consists of the objectives (KPMs), the language in which they are coded, and the optimization method and overall parameters. The right column is an interactive display of the architecture along with the button to begin optimization, a progress bar, and controls to page through solutions.

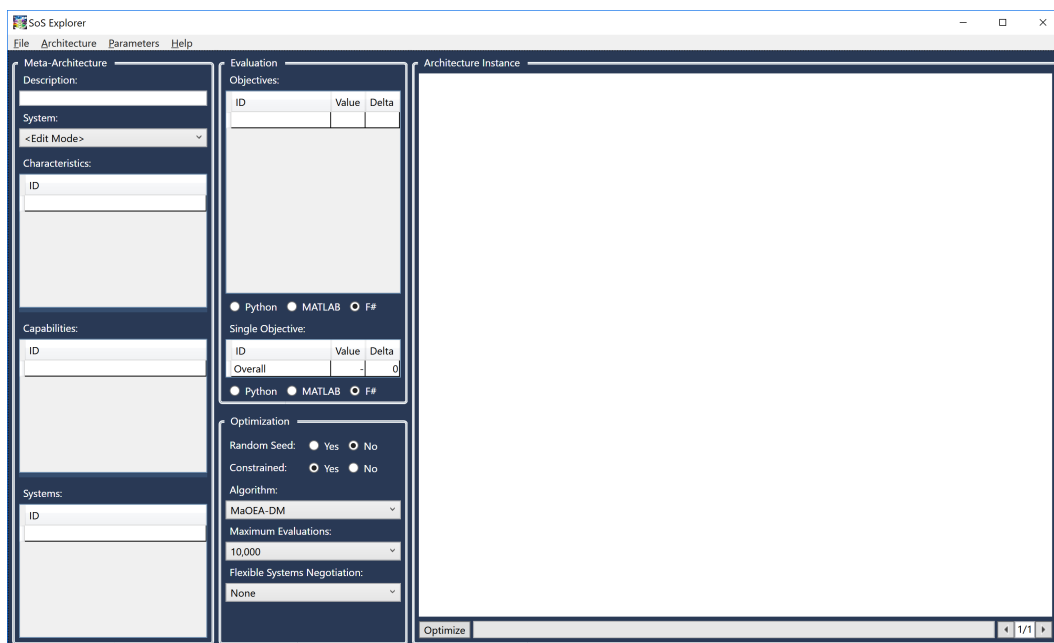


Figure 1: SoS Explorer when first opened.

In order to learn how to use SoS Explorer, let's work through a very simple problem involving a communications network.

B Communications network problem

B.1 Problem definition

In this problem, we are attempting to optimize a small communications network. The network requires two capabilities: voice and data. The KPMs (objectives) are affordability, voice coverage, and data coverage, all of which are to be maximized. These will be calculated based upon cost and range, so these will be the characteristics. There are six systems available for the network: four different transmitters and two different repeaters. The characteristics and capabilities of each system are summarized in Table 1 while the feasible interfaces are found in Table 2.

Table 1: Communications network system attributes

System	Cost (\$1,000's)	Range (km)	Voice	Data
Transmitter Type A	10	2	✓	
Transmitter Type B	20	2		✓
Transmitter Type C	40	5	✓	
Transmitter Type D	80	5		✓
Repeater 1	30	3		
Repeater 2	50	4		

Table 2: Communications network feasible interfaces

System	Type A	Type B	Type C	Type D	Repeater 1	Repeater 2
Transmitter Type A					✓	✓
Transmitter Type B					✓	✓
Transmitter Type C					✓	✓
Transmitter Type D					✓	✓
Repeater 1	✓	✓	✓	✓	✓	✓
Repeater 2	✓	✓	✓	✓	✓	✓

In order to be feasible, the following constraints must be maintained:

- A network must have both voice and data capability.
- Transmitters may not interface with one another.
- A repeater may have at most one transmitter with which it is interfaced.
- A chain of repeaters may have at most one transmitter interfaced any where in the chain.

Parts of these constraints are handled implicitly through the feasible interfaces defined for each system. However, others must be enforced explicitly which will be covered later.

B.2 Modeling the architecture

Now it is time to enter this data into SoS Explorer. First enter the description, characteristic names, capability names, system names, and objective names as shown in Figure 2. Next enter the associated data as shown in Figure 3 by selecting each system individually from the system drop-down box. Notice that where the system names were entered is now where the feasible interfaces are entered. If you need to add, modify, or delete any of the systems, characteristics, or capabilities, select <Edit Mode> from the system drop-down box. After you're finished, save the project—preferably in its own folder because the next step will create an associated sub-folder.

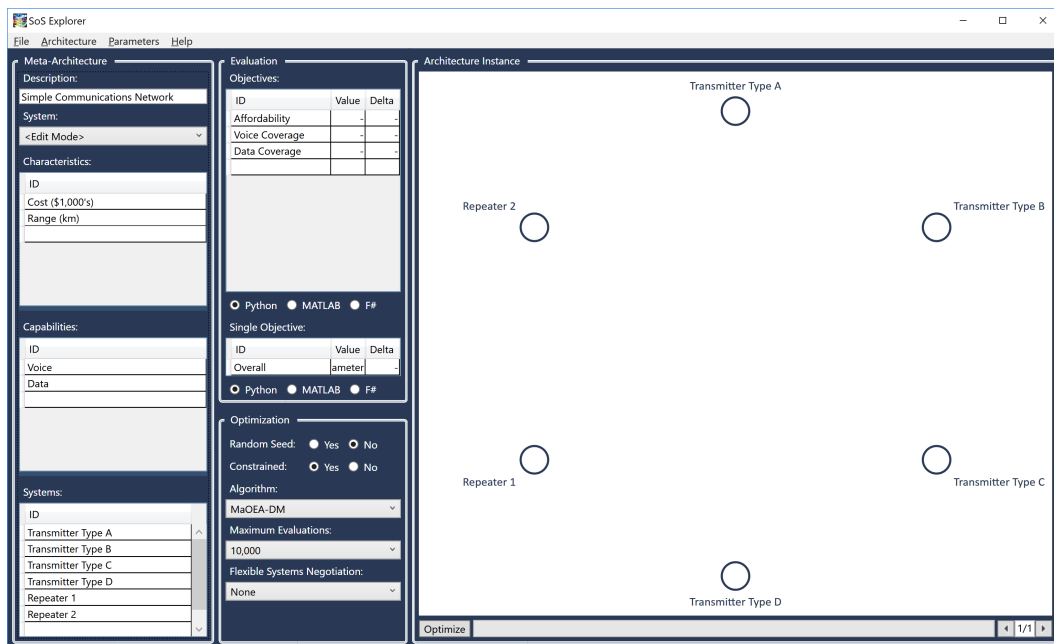


Figure 2: Enter names of systems, characteristics, capabilities, and objectives.

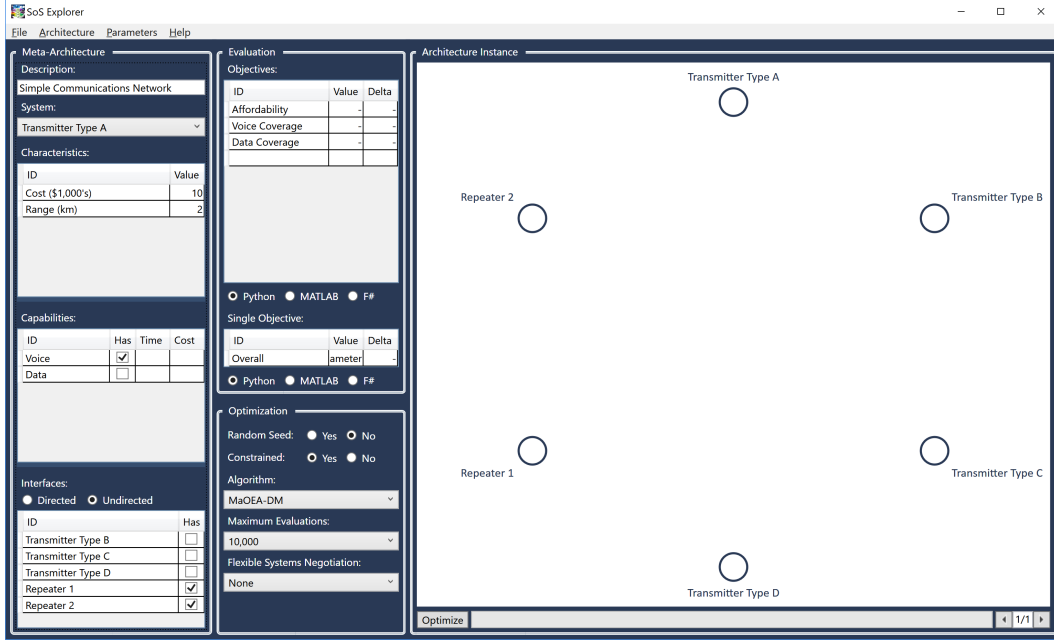


Figure 3: Enter data for each system.

B.3 Modeling the objectives

The problem is modeled through its objectives and a set of constraints. The default constraints require that solutions include each required capability and exclude any infeasible interfaces. These can be modified in the same way as the objectives and will be addressed later. The objectives may be modeled using Python, MATLAB, or F#. For this example, Python will be used because it comes with SoS Explorer while the others must be installed separately (F# is free from Microsoft). In SoS Explorer, select both of the Python radio buttons in order to evaluate the objectives using Python. Next choose File→Create Python Files. This will create a number of Python files in a sub-folder named “Python”. These are template files designed to assist in modeling the objectives and constraints. The objectives are in the files beginning with Objective_.

In order to model the objectives, the following notation will be employed: the characteristics matrix, (\mathbf{C}) , has dimensions $N_C \times N_S$ where N_C is the number of characteristics and N_S is the number of systems. \mathbf{C}_{ij} is defined as the i th characteristic of the j th system. Likewise, the capabilities matrix, (\mathbf{C}') , has dimension $N_{C'} \times N_S$ where $N_{C'}$ is the number of capabilities. Finally, the feasible interface matrix, (\mathbf{F}) , has dimension $N_S \times N_S$.

It is important to note that SoS Explorer uses evolutionary algorithms for optimization. Evolutionary algorithms represent solutions, in this case architectures, as chromosomes. Therefore, the objectives take as parameters a chromosome, characteristics, capabilities, feasible interfaces, and some others beyond discussion here. The chromosome contains which systems and interfaces are selected and nothing more. It is helpful to introduce functions representing this: the functions S and I extract the system and interface information from a chromosome and are defined as

$$S(\mathbf{X}, i) = \begin{cases} 1 & \text{if the } i\text{th system is selected in } \mathbf{X} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

and

$$I(\mathbf{X}, i, j) = \begin{cases} 1 & \text{if the } i\text{th and } j\text{th systems have an interface in } \mathbf{X} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where \mathbf{X} is the chromosome.

First let's model affordability (O_1). For this, we will use the formula

$$O_1(\mathbf{X}, \mathbf{C}) = - \sum_{i=1}^{N_S} S(\mathbf{X}, i) C_{\text{Cost}, i} \quad (3)$$

which is simply the negative of the sum of the costs of all selected systems.

To implement this in Python, open the `Objective_Affordability.py` file in a text editor. You will notice that there are indices defined for each of the systems, characteristics, and capabilities along with a function header and the helper functions `hasSystem` and `hasInterface`. These should not be modified. The calculation should replace all of the code found following the comment `Replace below with actual objective calculation!`. Use the following code for this calculation:

```
# Add up the cost of each participating system
totalCost = 0.0
for i in range(0, numSystems):
    if hasSystem(i):
        cost = characteristics[i, char_Cost1000s]
        totalCost += cost

# Return results
return -totalCost
```

The second objective, voice coverage (O_2), can be modeled using

$$O_2(\mathbf{X}, \mathbf{C}, \mathbf{C}') = \pi \sum_{i=1}^{N_S} S(\mathbf{X}, i) V(\mathbf{C}', i) \left[C_{\text{Range}, i}^2 + \sum_{\substack{j=1 \\ j \neq i}}^{N_S} S(\mathbf{X}, j) I(\mathbf{X}, i, j) R(\mathbf{C}', j) \Gamma(\mathbf{C}', i, j) \left(C_{\text{Range}, j}^2 + \sum_{\substack{k=1 \\ k \neq i, j}}^{N_S} S(\mathbf{X}, k) I(\mathbf{X}, j, k) R(\mathbf{C}', k) \Gamma(\mathbf{C}', j, k) C_{\text{Range}, k}^2 \right) \right] \quad (4)$$

where

$$V(\mathbf{C}', i) = \begin{cases} 1 & \text{if the } i\text{th system is a voice transmitter (voice in } \mathbf{C}') \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

$$R(\mathbf{C}', i) = \begin{cases} 1 & \text{if the } i\text{th system is a repeater (no data or voice in } \mathbf{C}') \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

and a loss factor

$$\Gamma(\mathbf{C}', i, j) = \begin{cases} 0.5 & \text{if } R(\mathbf{C}', i) > R(\mathbf{C}', j) \\ 0.7 & \text{otherwise} \end{cases} \quad (7)$$

This is roughly the circular area covered by the range of each selected transmitter augmented by each selected repeater (with a loss factor for overlap) that is interfaced to it. Repeater chains are assumed to have a maximum of two repeaters. The following code implements this calculation:

```
# Define isVoiceTransmitter() function
# Returns whether the given system is a voice transmitter
def isVoiceTransmitter(i):
    hasVoice = capabilities[i, cap_Voice]
    return hasVoice

# Define isRepeater() function
# Returns whether the given system is a repeater
def isRepeater(i):
    hasVoice = capabilities[i, cap_Voice]
    hasData = capabilities[i, cap_Data]
    return not (hasVoice or hasData)

# Add up coverage area for each participating system
coverageArea = 0.0
for i in range(0, numSystems):
    if hasSystem(i) and isVoiceTransmitter(i):
        rangeSystem1 = characteristics[i, char_Rangekm]
        coverageArea += rangeSystem1 * rangeSystem1
    for j in range(0, numSystems):
        if hasSystem(j) and isRepeater(j) and hasInterface(i, j):
            rangeSystem2 = characteristics[j, char_Rangekm]
            if rangeSystem1 > rangeSystem2:
                coverageArea += 0.5 * rangeSystem2 * rangeSystem2
            else:
                coverageArea += 0.7 * rangeSystem2 * rangeSystem2
    for k in range(0, numSystems):
        if hasSystem(k) and isRepeater(k) and hasInterface(j, k):
            rangeSystem3 = characteristics[k, char_Rangekm]
            if rangeSystem2 > rangeSystem3:
                coverageArea += 0.5 * rangeSystem3 * rangeSystem3
            else:
                coverageArea += 0.7 * rangeSystem3 * rangeSystem3

# Return results
return 3.14 * coverageArea
```

The third objective, data coverage (O_3), is handled in a similar fashion using

$$O_3(\mathbf{X}, \mathbf{C}, \mathbf{C}') = \pi \sum_{i=1}^{N_S} S(\mathbf{X}, i) D(\mathbf{C}', i) \left[C_{\text{Range}, i}^2 + \sum_{\substack{j=1 \\ j \neq i}}^{N_S} S(\mathbf{X}, j) I(\mathbf{X}, i, j) R(\mathbf{C}', j) \Gamma(\mathbf{C}', i, j) \left(C_{\text{Range}, j}^2 + \sum_{\substack{k=1 \\ k \neq i, j}}^{N_S} S(\mathbf{X}, k) I(\mathbf{X}, j, k) R(\mathbf{C}', k) \Gamma(\mathbf{C}', j, k) C_{\text{Range}, k}^2 \right) \right] \quad (8)$$

where

$$D(\mathbf{C}', i) = \begin{cases} 1 & \text{if the } i\text{th system is a data transmitter (data in } \mathbf{C}') \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

The following code implements this calculation:

```
# Define isDataTransmitter() function
# Returns whether the given system is a data transmitter
def isDataTransmitter(i):
    hasData = capabilities[i, cap_Data]
    return hasData

# Define isRepeater() function
# Returns whether the given system is a repeater
def isRepeater(i):
    hasVoice = capabilities[i, cap_Voice]
    hasData = capabilities[i, cap_Data]
    return not (hasVoice or hasData)

# Add up coverage area for each participating system
coverageArea = 0.0
for i in range(0, numSystems):
    if hasSystem(i) and isDataTransmitter(i):
        rangeSystem1 = characteristics[i, char_Rangekm]
        coverageArea += rangeSystem1 * rangeSystem1
        for j in range(0, numSystems):
            if hasSystem(j) and isRepeater(j) and hasInterface(i, j):
                rangeSystem2 = characteristics[j, char_Rangekm]
                if rangeSystem1 > rangeSystem2:
                    coverageArea += 0.5 * rangeSystem2 * rangeSystem2
                else:
                    coverageArea += 0.7 * rangeSystem2 * rangeSystem2
            for k in range(0, numSystems):
                if hasSystem(k) and isRepeater(k) and hasInterface(j, k):
                    rangeSystem3 = characteristics[k, char_Rangekm]
                    if rangeSystem2 > rangeSystem3:
                        coverageArea += 0.5 * rangeSystem3 * rangeSystem3
                    else:
```

```
coverageArea += 0.7 * rangeSystem3 * rangeSystem3

# Return results
return 3.14 * coverageArea
```

Now that the objectives are modeled, SoS Explorer can evaluate architectures. The SoS Explorer objectives should now have numeric values associated with them as in Figure 4. You can right-click on nodes to select whether or not a system is participating and left-click and drag between nodes to create interfaces. Interfaces may be removed by right-clicking on them and selecting “Remove”. Left-clicking an empty area allows for a note to be added or edited. However, the end-goal is to have the computer do the heavy-lifting, so the evolutionary algorithms will generate the architectures for us. These architectures are still editable, and it is easy to make copies of architectures, delete them, remove duplicates, and many other operations using the “Architecture” menu. Also, SoS Explorer allows multiple solutions to be on hand with paging buttons in the lower right corner. In order to allow the optimizer to create feasible architectures, there is one step remaining: adding constraints.

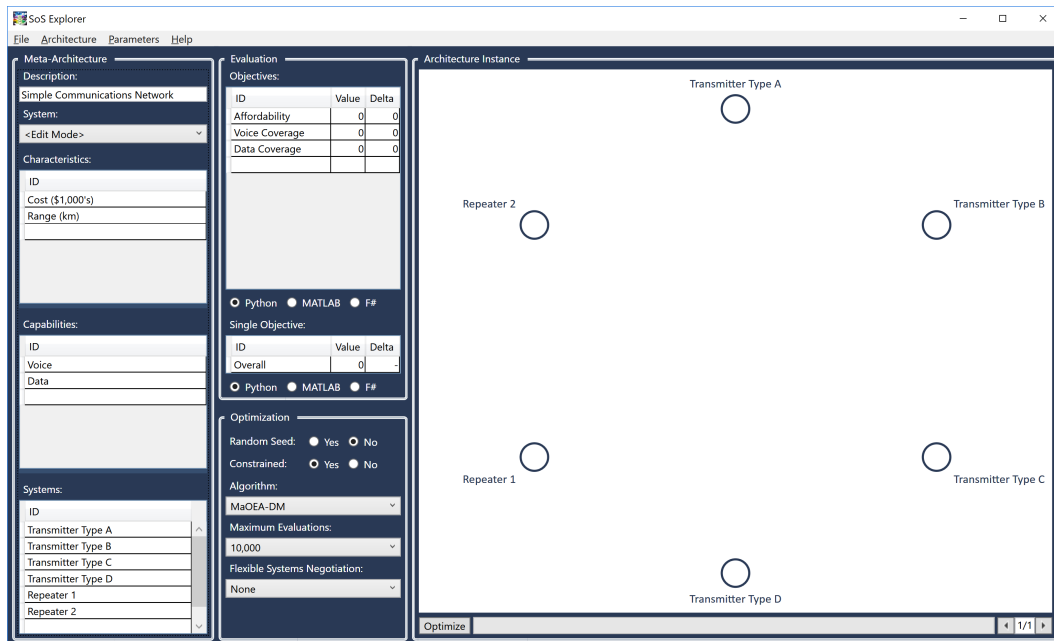


Figure 4: Objectives are now evaluated.

B.4 Modeling the constraints

The evolutionary algorithms allow constraints to be enforced via a mechanism known as chromosome fixing. Using this technique, the chromosomes are modified to meet feasibility requirements. An issue with using fixing is that it can work against the evolutionary process. To mitigate this, the actual chromosomes are not modified, but rather a function, \mathbf{G} , is used to map the chromosome to its feasible complement which is then passed into the objective function. In other words, when constraints are enabled then the objectives are passed $\mathbf{G}(\mathbf{X})$ instead of \mathbf{X} . This way, the evolutionary operations are not undermined by the enforcing of constraints. The file where the constraints are modeled is `FixChromosome.py`.

There are four constraints required by this problem. The first two constraints, both voice and data capabilities and no infeasible interfaces are already performed by the template code. The other two must be added. The following code may be added to the end of the following in order to require that no more than one transmitter per repeater and no more than one transmitter per repeater chain:

```
# Define isRepeater() function
# Returns whether the given system is a repeater
def isRepeater(i):
    hasVoice = capabilities[i, cap_Voice]
    hasData = capabilities[i, cap_Data]
    return not (hasVoice or hasData)

# Define hasTransmitter() function
# Returns whether the given system is interfaced to a transmitter
def hasTransmitter(i):
    for j in range(0, numSystems):
        # If different systems with an interface and transmitter
        if i != j and hasInterface(i, j) and isTransmitter(j):
            return True
    return False

# Limit repeaters to one transmitter interface
for i in range(0, numSystems):
    if isRepeater(i):
        count = 0
        for j in range(0, numSystems):
            # If different systems with an interface and transmitter
            if i != j and hasInterface(i, j) and isTransmitter(j):
                # Limit to one transmitter per repeater
                count += 1
                if count > 1:
                    setInterface(i, j, False)

# Limit repeaters to one transmitter in chain
for i in range(0, numSystems):
    if isRepeater(i) and hasTransmitter(i):
        for j in range(0, numSystems):
            # If different systems with an interface
            if i != j and hasInterface(i, j):
                # Limit to one transmitter per repeater
                if isRepeater(j):
                    for k in range(0, numSystems):
                        # If different systems with an interface
                        if i != k and j != k and hasInterface(j, k):
                            if isTransmitter(k):
                                setInterface(j, k, False)
```

After this is added, we are ready to run the optimizer.

C Optimization

There are two basic categories of optimization: single objective and multiple objective. Multiple objective is actually simpler to use because it doesn't require a function to combine the given objective into a single overall objective. Therefore, we will first tackle multiple objective optimization.

C.1 Multiple objective optimization

First make sure that “MaOEA-DM” is selected for the algorithm, maximum evaluations is set to 10,000, random seeding is off, constraints are on, and that flexible systems negotiation is “None”. Next, select Parameters→MaOEA-DM and enter the data as shown in Figure 5.

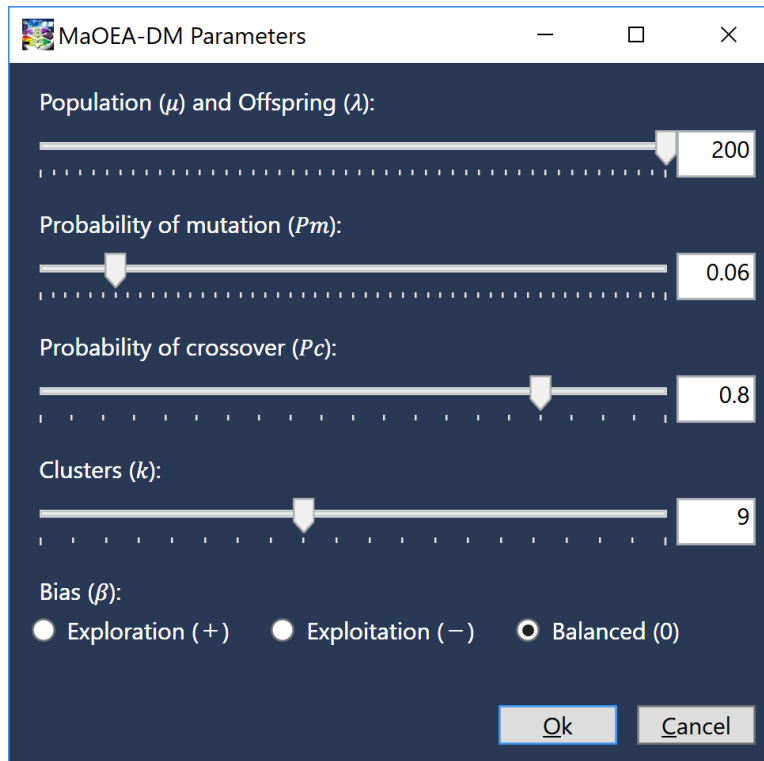


Figure 5: MaOEA-DM parameters menu.

Finally, you're ready to optimize. Click the “Optimize” button and wait until the process is finished. Everything is well is you get six results where the first and fifth match Figure 6 and Figure 7, respectively.

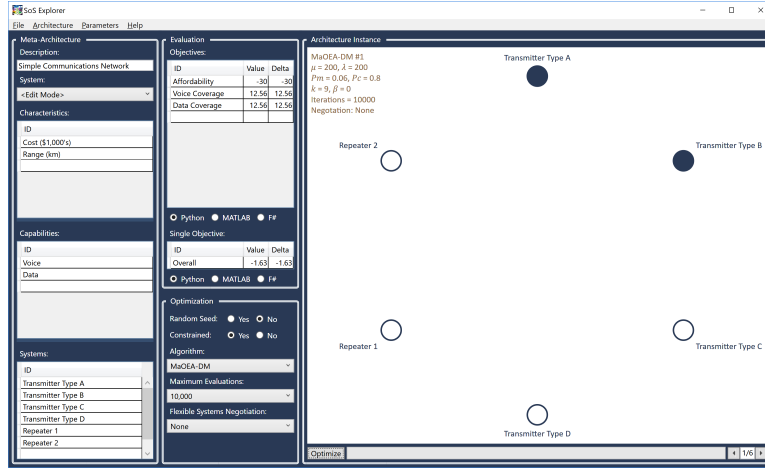


Figure 6: Solution number 1.

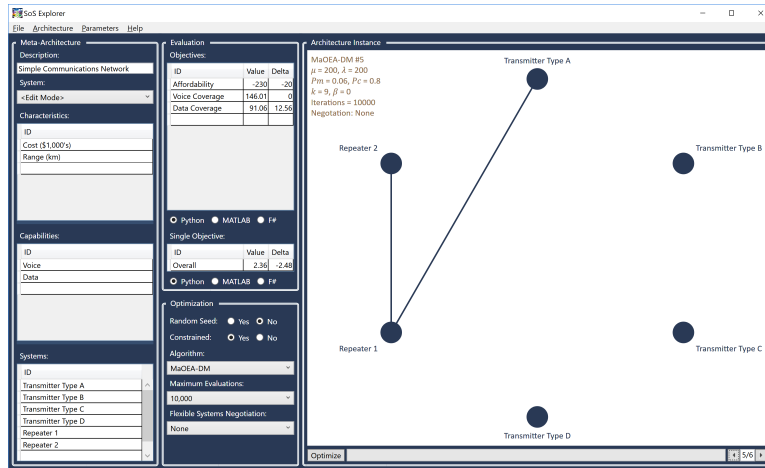


Figure 7: Solution number 5.

These first solution represents the most affordable architecture and the second represents the highest performing. The rest are other solutions belonging to the optimal (Pareto) set. By paging through them, the deltas next to the objective values let you know how the values are changing. You have now finished your first project in SoS Explorer!

C.2 Single objective optimization

Now for single objective optimization. In this case a function must be defined that maps the given objectives to a single value that represents the overall objective. This function can be created using a simple weighting scheme or nonlinear methods such as a fuzzy associative memory. The overall objective function, O_{Overall} , therefore has the form

$$O_{\text{Overall}} : \mathbb{R}^{N_O} \mapsto \mathbb{R} \quad (10)$$

where N_O is the number of objectives.

The Python and F# template code implements a simple weighting scheme while the MATLAB template code incorporates a fuzzy inference system (requires the MATLAB Fuzzy Logic Toolbox) that can be edited through SoS Explorer by selecting File→Edit FIS File. Restricting the discussion to Python for this introduction, the file where the overall objective is implemented is Overall_Objective.py. As an exercise, let's modify the code to weight the objectives as follows: affordability at 25%, voice coverage at 5%, and data coverage at 70%. Replacing the code after the Replace below with actual objective calculation! comment block with the following will achieve the desired result:

```
# Defines weights for each objective
w_Affordability = 0.25
w_VoiceCoverage = 0.05
w_DataCoverage = 0.70

# Calculate weighted overall objective
overall = w_Affordability * objectives[obj_Affordability] + \
          w_VoiceCoverage * objectives[obj_VoiceCoverage] + \
          w_DataCoverage * objectives[obj_DataCoverage]

# Return overall objective
return overall
```

There is one single objective optimizer, “Simple SOGA”, included with SoS Explorer. Select it from the algorithm drop-down box and then click the optimize button. Since the overall objective is heavily weighted towards data, it would be expected that data coverage will be maximized and that voice coverage will be minimal since it is weighted less than affordability. The results are exactly as expected as shown in Figure 8.

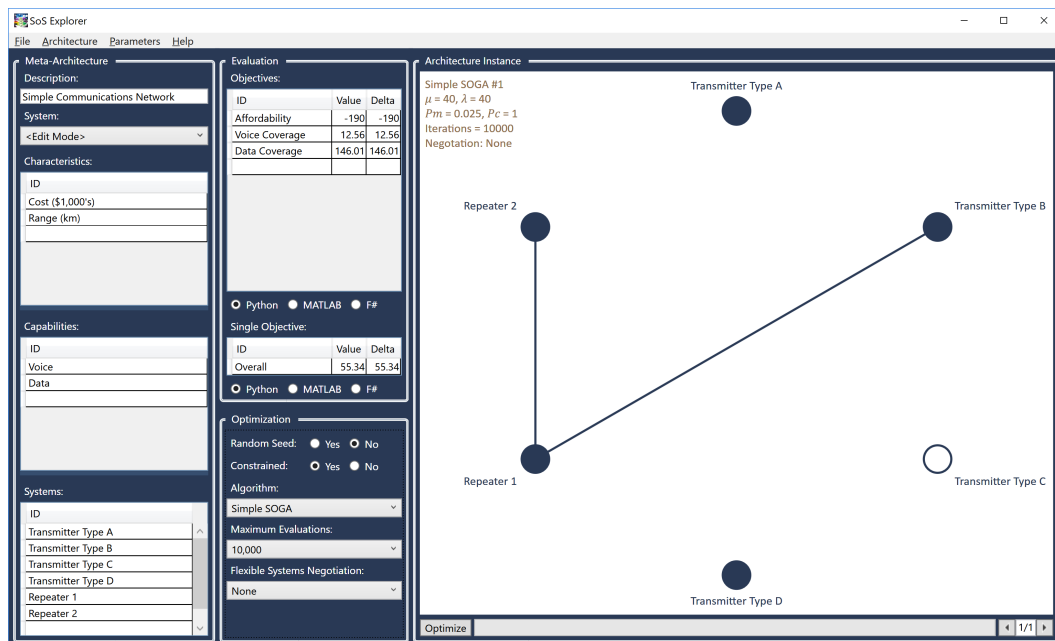


Figure 8: Single objective solution.

This is the end of this introductory tutorial. SoS Explorer has other capabilities such as flexible

systems, but these will be addresses in a separate tutorial.