DPA configuration in MCP

Author: sergebdt@google.com - Google
Version: 1.0

This document describes how to configure the E-DPAs in the MCP config file.
See as an example the provided MCP1 config file.

The DPA declaration in the config file

You will notice first a declaration of the DPAs used by the test:

```
 "dpas": {
    "East4": {
      "movelistMargin": 10,
      "points_builder": "default (25, 10, 10, 10)"
    },
    "East5": {
      "movelistMargin": 10,
      "points_builder": "default (25, 10, 10, 10)"
    },
    "East6": {
      "movelistMargin": 10,
      "points_builder": "default (25, 10, 10, 10)"
    }
  },
```

Here we declare each E-DPA to be simulated by the test harness, by using their official name (*East4*, ...) as defined in the official E-DPA.kml file that you can find in the winnforum repository here.

For each DPA there are 2 parameters set:

- `movelistMargin`: the margin used for doing the aggregate interference test.
- `points_builder`: this is where you define the method to generate the DPA protected points. The way to use that is described more precisely in second section of this document.

Later on in the config file, you have instruction for DPA activation and deactivation as part of the MCP cycles, for example:

```
"dpaActivationList": [
     {
       "dpaId": "East4",
       "frequencyRange": {
         "lowFrequency": 3550000000
         "highFrequency": 3650000000,
       }
     },
     {
       "dpaId": "East5",
       "frequencyRange": {
         "lowFrequency": 3550000000
         "highFrequency": 3650000000,
       }
     }
```

or for deactivation:

```
"dpaDeactivationList": [
     {
       "dpaId": "East4",
       "frequencyRange": {
         "lowFrequency": 3550000000
         "highFrequency": 3650000000,
       }
     }
```

On those you simply specify which DPAs and frequency range to activate/deactivate at that stage of the MCP process.

How to configure the DPA protected points

The "`points_builder`" directive above is where you specify this.
It supports three methods:

A. using the default builder which automtically generates the points for you.
B. specifying a GeoJSON file of type `` `MultiPoints` `` listing all the points to use.
C. defining your own "*builder*" routine.

In practice the MCP test actually calls the `BuildDpa`() routine [found in `dpa_mgr.py`].

Method A: using the default builder

Simply specify the string "default <parameters>" with the <parameters> that you want.
For example:

```
"points_builder": "default (25, 10, 10, 10)"
```

As explained in the [README.md file]:
Provides a simple `` `default` `` builder that performs a distribution of points within the DPA with a simple strategy:

- divide the DPA in 4 entities: front border, back_border, front zone, back zone
- distribute points along the 2 borders in a linear fashion, and within the 2 zones in a grid fashion. The number of points can be specified for each.
- the division of front to back is done by cutting the DPA with an expanded US border. Obviously inland portal DPAs only contains a front border and front zone. The expansion parameter is configurable. By default it is 40km.
- a minimum distance can be specified between protected points of each of these 4 entities separately. By default 0.2km for front border, 0.5km for front zone, 1km for front zone and 3km for back zone. Note that the distance are approximative.

The total number of points distributed is variable depending on these condition. Common strategies can be:

- define a large maximum number of points for each of the entities and define a realistic distance between points ⇒ the zones will be gridded by distance only.
- define the required number of points, and a minimum distance to insure than small DPAs will not have too many points very close from each other.

Here is also the description from the `BuildDpa()` routine documentation:

'`default <parameters>`': A simple default method. Parameters is a tuple defining the number of points to use for different part of the DPA:

`num_pts_front_border`: Number of points in the front border
`num_pts_back_border`: Number of points in the back border
`num_pts_front_zone`: Number of points in the front zone
`num_pts_back_zone`: Number of points in the back zone
`front_us_border_buffer_km`: Buffering of US border for delimiting front/back.
`min_dist_front_border_pts_km`: Minimum distance between front border points (km).
`min_dist_back_border_pts_km`: Minimum distance between back border points (km).
`min_dist_front_zone_pts_km`: Minimum distance between front zone points (km).
`min_dist_back_zone_pts_km`: Minimum distance between back zone points (km).

Example of encoding: `default (200,50,20,5)`

Notes:

+ the default values are (25, 10, 10, 5, 40, 0.2, 1, 0.5, 3)
+ only the parameters defined here will be different from default values, the one no defined will use the default value
+ the result are only approximate (actual distance and number of points may differ slightly).

Method B: using a GeoJson file of type 'MultiPoints'

Simply specify a path to your GeoJson file. The path can be either absolute, or relative to the running script (usually the '`harness/`' directory, if for example you are using the '`harness/test_main.py`' script).

For example:

```
"points_builder": "/path/to/my/config.json"
```

A GeoJSON MultiPoints file can be easily created. For example let's suppose you have 2 points that you want to use for **DPA East5**.

Then you can simply do in python:

```
import shapely.geometry as sgeo
from reference_models.geo import utils

pt1 = (-69.58, 43.88)   ## (longitude, latitude) format
pt2 = (-69.10, 43.20)
points = sgeo.MultiPoints([pt1, pt2])
json_str = utils.ToGeoJson(points)
with open('myfile.json', 'w') as fd:
        fd.write(json_str)
```

Notes:
+ the points needs to be contained by the DPA. The test harness code will verify this and raise an exception if it is not the case.

Method C: defining your own builder

This method is more involved, as you need to provide your own builder routine.
First define your own builder routine:

```
def my_builder_fn(dpa_name, dpa_geometry, arg1, arg2, arg3, ...):
  """Generates the protected points for a given DPA.

    See the file reference_models/dpa/dpa_builder.py.

  Args
    dpa_geometry: A shapely geometry defining the DPA, as a shapely
                  Polygon, MultiPolygon or Point.
    arg1, arg2, ...: the list of parameters that your routine need.
                  Any number of arguments can be defined.
  """
  ... do something here to build the points for that DPA...
  ... using the dpa_geometry and the parameters ...
```

In your main script launching the test harness, add the following to register your builder:

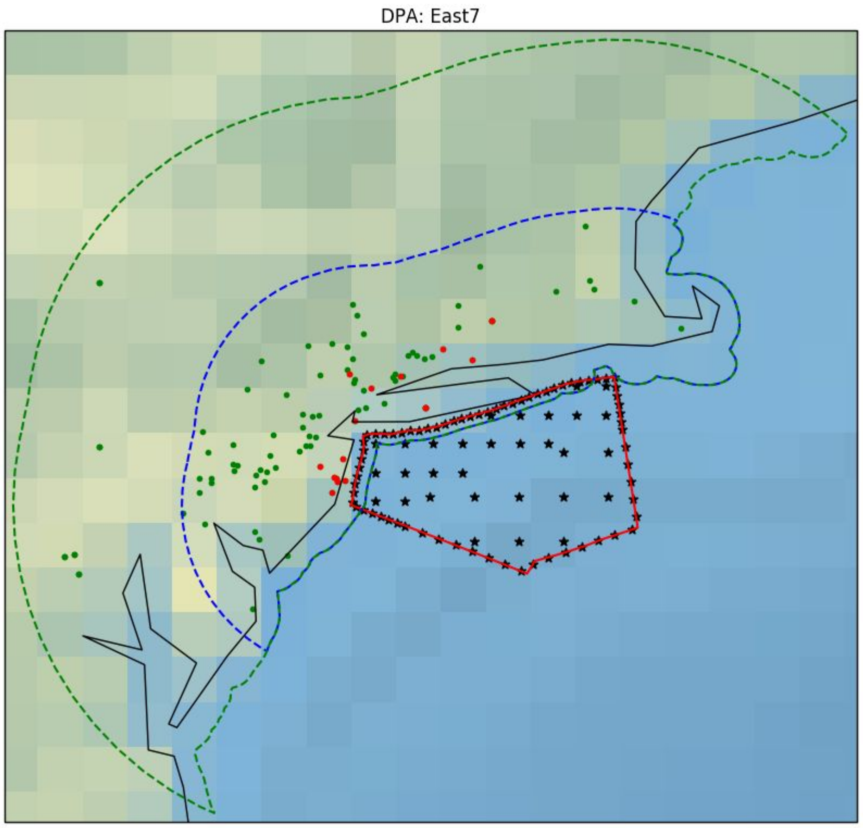
```
    from reference_models.dpa import dpa_builder
    dpa_builder.RegisterMethod('my_builder', my_builder_fn)
```

Now in the MCP config file you simply have to specify:

```
   "points_builder": "my_builder (3, 5, 7)"
```

Example of Method A output

As an example of what the `default` builder of Method A is doing, you can check the result of the `time_dpa.py` too found here:



We see here the DPA in red, and the protected points within the DPA.
It has been produced using option:

```
default (50, 20, 20, 10, 40)
```

For generating about 100 points total (default interpoint distance used).

We can observe that the front border has a lot of points close from each other. The back border points have more separation. The inside zone is gridded at two different resolutions for the front zone and back zone. The seaparation between the front and back areas is about 40km from the US border KML file (not shown here).

Further recommendations

The MCP test (as well as some other functional tests) are calculation heavy on the test harness. For best performance, It is particularly important to properly configure:

- the multiprocessing, so that the heavy calculation can be done in parallel. Multiprocessing is mostly used when computing PPA, GWPZ and DPA. A good rule of thumb is to use a ratio of the number of cpu-core available in your machine, for example 50% or 80%.
- The size of the geo terrain memory cache. For efficiency the terrain and NLCD drivers maintain a tile cache in LRU (Least recently used) fashion. The cache allows to avoid the performance hit of reloading tiles in memory when doing repetitive propagation calls in the same geographical area.

The provided main script (file `test_main.py`) implements an automatic setup for those two things, by using a strategy for allocating a portion of the available RAM to the two caches (terrain and NLCD) knowing the expected use of each. See the comments on that file for an explanation.

It is recommended to tune this automatic setup for your environment (or port a similar setup to your launch script if not using the provided `test_main.py`).
See in particular the set of control variables:

```
# Multi-processing
# Number of worker processes allocated for heavy duty calculation,
# typically DPA, PPA creation and IAP for PPA/GWPZ.
# Values:
#  -1: Use half of the cpu available
#  -2: Use all available cpu minus one (good idea to keep one cpu
core  #      for other computer usage).
#  Other: the number of process to allocate
NUM_PROCESSES = -2

# Memory allocation management
# A simple strategy is proposed by default, knowing the test harness
# usage:
#   - NLCD only used for PPA and GWPZ for deriving the Land Cover.
#     Area size is limited in general. LandCover extraction is done
in
#     main thread only (except for PPA tests where worker processes
#     check the CBSD landcover).
#     ==> Allocate most of the cache to main processes
#   - NED used by all others, with heavy terrain profile reading by
#     the main and worker threads.
#     ==> Allocate equal memory to worker processes, and twice to
main
#         process
# This strategy could be tuned for best performance.
#
# A 2 level cache could also be used (with memmap or otherwise) in
# order to share most of the in-memory tiles across processes. Not
# implemented.
#
# The memory allocated for geo cache across all sub-processes (in
MB).
#  -1: automatic allocation
#  otherwise this number (in MB) is used for setting up the cache
size
MEM_ALLOCATION_GEO_CACHE_MB = -1

# When 'automatic allocation', the ratio of total physical memory
# dedicated to the geo cache
MEM_RATIO_FOR_GEO_CACHE = 0.5
# The weighting factor of the main processes vs worker processes.
MEM_NED_WEIGHT_MASTER = 2.0
MEM_NLCD_WEIGHT_MASTER = 2.0
MEM_NLCD_CACHE_WORKERS = 6.0
```