Script Templating for EXOSIMS

Michael Turmon

2018 November 28

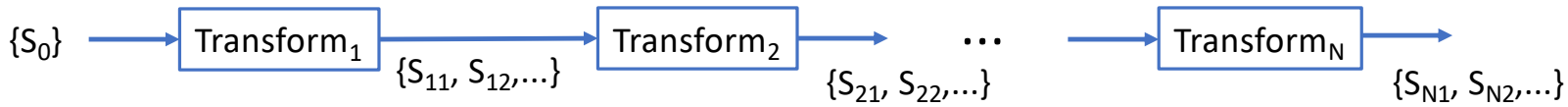
Goal

- Support generation of a family of derived scripts using one valid script as a starting point.
- Scenario: We have one script that works. We want to do parameter optimization or sensitivity studies on the basis of that script.
- Two worthwhile ideas here: compositional script transforms, and naming conventions.

- Thanks to: Dean Keithly for discussions and prior work ("makeSimilarScripts.py")

Basic Idea

- System takes one script as input, and at each of N stages, transforms that script into a set of scripts (script = dictionary).
- Envision the initial script as a one-script set. Then the transformation process looks like:



- At each stage, each input script gets turned in to one or more output scripts.
- All output scripts after a given stage are pooled, and given as input to the next stage.

Three Kinds of Transform

1. Update (1 input, 1 output):
   - The transform contains a dictionary of values. Update the values in the input dictionary, with those in the transform dictionary.
   - T = {”dMagLim”: 23} -> set dMagLim in all input scripts to 23.
2. Sequence (1 input, M outputs): **<== In practice, this is the most used.**
   - The transform contains a dictionary of values that are vectors. For each element in the vector, output a new script with that value.
   - T = {"dMagLim": [23, 24, 25]} -> make 3 output scripts, one with each value.
3. Cross product (1 input, M x P x ... outputs):
   - Transform contains a dictionary of (several) vectors, and all combinations of their elements are used to create output scripts.
   - T = {"dMagLim": [23, 24, 25], "minComp": [0, 0.1]} -> 6 output scripts are generated, with each combination of values.

Compositional

- The transforms can be composed as many times as desired.
- For instance, an original script can be left intact and one or more updates can be applied to it – transforming that one script into a different (single) script.
- Then a sequence of values can be plugged in to a different element in the script.
- As an example, two sequence transforms are equivalent to a cross-product (Cartesian product) transform that has two target elements.

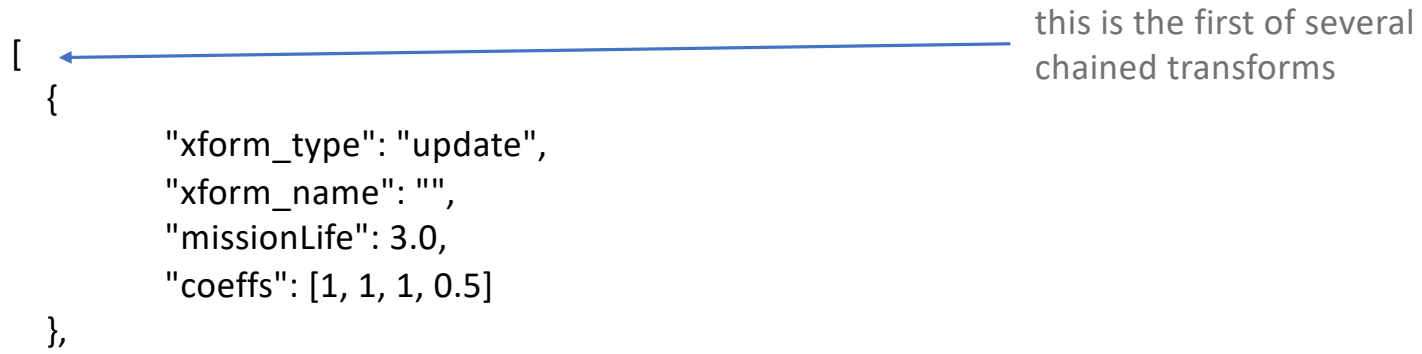
Naming

- *Naming sounds trivial but it is important to the system function.*
- We have to say which kind of transform we intend.
  - We use the “xform_type” attribute, right in the transform.
- We have to name the resulting script.
  - We use a “xform_name” attribute, also in the transform.
  - It uses the Python format language, so you can do clever things.
  - E.g.: xform_name = “c1={coeffs[2]:.2f}” -> use the %.f format on the third element of the coeffs array in the dictionary as this fragment of the name.
- The names are *chained*, same as the transforms: names accumulate by *concatenation* in sensible ways as the transforms are performed.

Example (1)

We want to iterate over 3 components of a four-element coefficient vector.

Start out with simple update, to reset the mission life and set up coeffs[3] = 0.5, which will not change.



Example (2, cont.)

Next, make a sequence over 6 choices for coeffs[0]. We go from 1 script input to 6 scripts output. Output filename example is “c1=0.10”.

```
{
        "xform_type": "sequence",
        "xform_name": "c1={coeffs[0]:.2f}",
        "xform_action": {"coeffs": "masked"},
        "coeffs": [[0.05, NaN, NaN, NaN],
                   [0.2, NaN, NaN, NaN],
                   [0.4, NaN, NaN, NaN],
                   [0.6, NaN, NaN, NaN],
                   [0.8, NaN, NaN, NaN],
                   [1.0, NaN, NaN, NaN]
                  ]
  },
```

masked = NaNs do not over-ride extant values

Example (3)

Now, 6 further choices for coeffs[1]. Six scripts input, 36 output. Output filename example is “c1=0.10_c2=0.80” because names accumulate by concatenation.

```
{
        "xform_type": "sequence",
        "xform_name": "c2={coeffs[1]:.2f}",
        "xform_action": {"coeffs": "masked"},
        "coeffs": [[NaN, 0.05,NaN, NaN],
                   [NaN, 0.2, NaN, NaN],
                   [NaN, 0.4, NaN, NaN],
                   [NaN, 0.6, NaN, NaN],
                   [NaN, 0.8, NaN, NaN],
                   [NaN, 1.0, NaN, NaN]
                  ]
  },
```

Example (4)

6 choices for coeffs[2]. 36 in, 216 out.
Output filename example is “c1=0.10_c2=0.80_c3=1.00”

Note: Chaining all previous scripts gives a complete sweep, in one file.

```
{
        "xform_type": "sequence",
        "xform_name": "c3={coeffs[2]:.2f}",
        "xform_action": {"coeffs": "masked"},
        "coeffs": [[NaN, NaN, 0.05,NaN],
                   [NaN, NaN, 0.2, NaN],
                   [NaN, NaN, 0.4, NaN],
                   [NaN, NaN, 0.6, NaN],
                   [NaN, NaN, 0.8, NaN],
                   [NaN, NaN, 1.0, NaN]
                  ]
} ]
```

this is the last of the chain of transforms

More On Naming

- We also found it useful to extend the name syntax a bit.
- The driving need was nested parameters, especially modules. We wanted to be able to *update submodules* without *over-writing* all other choices.

```
{
 "xform_type": "update",
 "+modules": {
  "SimulatedUniverse": "SAG13Universe",
  "SurveyEnsemble": "IPClusterEnsemble"
 }
},
```

```
{
 "xform_type": "update",
 "modules": {
  "SimulatedUniverse": "SAG13Universe",
  "SurveyEnsemble": "IPClusterEnsemble"
 }
},
```

- "+" means "go inside and update" (for dictionaries) and "add to end" (for lists, e.g. scienceInstruments).

Realization: A Python script

```
# json-xform: Transform an EXOSIMS JSON script into a set of such scripts.
#
# Usage:
#   json-xform [-v] [-o FILE] SCRIPT XSCRIPT ...
#
# where:
#   SCRIPT is the JSON script used by EXOSIMS
#   XSCRIPT is a JSON script giving the transform
# and, optionally:
#   -o FILE to specify where files should go; FILE must
#      have one %s to receive a file index.
#   -v for some verbose output
```

Remarks

- I implemented a relative mode for all operations. It is kind of like "masked", in that it interprets the given values specially. In this case, as multipliers rather than as absolute values.

- This could use some kind of filter operation, so that some scripts could be eliminated. This would seem to require a programmatic interface (a predicate that would indicate what should be filtered), which I didn't see a clean/general way to do in JSON.

- In practice, we used the "update" and "sequence" transforms the most. We would typically use an offline procedure to generate a sequence of parameter values (sometimes nonuniform spacing was best), and then make the sequence into an input to the "json-xform" script.