2018-03-13 Antha, Transcriptic, AutoLims, & other computable workflow tools/apps/standards

compare "**projects**", "**types**," "**operations**," "**protocols**," "**runs**," "**units**", "**data**" in Antha, autoprotocol, Autodesk Wet Lab Accelerator, aquarium, Raik's rotmic lims.

TODO

- respond to https://github.com/scottbecker/autolims/issues/2#issuecomment-372840007
- summarize architecture; org notes on autoprotocol / antha / autolims / aquarium
  - emphasis on extensible & shareable Types and "op" data standard that wraps arbitrary documentation / protocols / workflows with metadata defining conceptual "function signature" based on said Types
  - goal is to enable modular, composable human-centric workflow design and sharing w/ data
  - not strictly computable (i.e. execution will require human in the loop initially)
  - iff traction, then deeper integration into / on top of existing computable workflow languages / standards.s
- respond again to scottbecker
- "walking skeleton" of typseschema + forms + notes
- connect w/ Brian Naughton / BooleanBiotech http://www.hexagonbio.com

---

jupyter for protocols & labwork

- https://github.com/dacarlin/robot-bagel
- http://nbviewer.jupyter.org/github/BjornFJohansson/ypk-xylose-pathways/blob/master/index.ipynb
- http://autoprotocol-python.readthedocs.io/en/latest/protocol.html#protocol-as-dict
- https://github.com/autoprotocol/autoprotocol-python/blob/master/autoprotocol/protocol.py

---

Autoprotocol & Transcriptic

autoprotocol: https://github.com/autoprotocol/autoprotocol-python/blob/master/autoprotocol/protocol.py

transcriptic: https://github.com/transcriptic/transcriptic/blob/master/transcriptic/english.py#L43

thoughts: autoprotocol.org/specification/#protocols identifies key parameters for 19 autoprotocol `ops` or `instructions`. Didact protocol metadata should be able to model these. Port them if licensing allows. Actual code for `ops` is defined in transcriptic/english.py.

autoprotocol "in the wild":

- "simpler" example of autoprotocol https://github.com/scottbecker/transcriptic101/blob/master/Transcriptic 101.ipynb
- great example of an autoprotocol metaprotocol http://blog.booleanbiotech.com/puc19_pcr_amplification.html

core objects jupyter module (docs; code)

- **project**: manages runs
- **run**: manage Instructions, Datasets, "monitoring data"
- **container**: container Type from the Transcriptic LIMS & aliquots present in the container
- **aliquots**: DataFrame of aliquots in the container, along with aliquot name, volume, and properties
- **dataset / data**: DataFrame of well-indexed data values. "*Note that associated metadata is found in attributes dictionary*" ??
- **job_tree**: "A Job Tree visualizes the instructions of a protocol in a hierarchical structure based on container dependency to help human readers with manual execution. Its construction utilizes the algorithm below, as well as the Node object class (to store relational information)"

key concepts:

refs
the set of containers that will be used in the protocol

instructions
Instructions are the unit operations of a protocol; the list of instructions to be performed

runs
A run is a specific instance of the execution of a given protocol. A run is composed of a sequence of instructions, which will be executed in order (with some parallelization where possible).

manifest.json
"A manifest.json file contains metadata about protocols required when uploading a package to Transcriptic. A package can contain many protocols but for our example it will contain just one. The "inputs" stanza defines expected parameter types which translate into the proper UI elements for that type when you upload the package to Transcriptic."
...
"It can be thought of as a markdown language you can use to create a graphical user interface for your protocols so that they can be parameterized and launched easily through the web app."

input types (manifest.json): https://developers.transcriptic.com/v1.0/docs/input-types
"An input refers to an editable field on a protocol within the protocol browser, they will allow you to replace the parameters you hard-coded into the preview once the protocol is uploaded as a package to the web app."

example:

```
# source
# blog.booleanbiotech.com/puc19_pcr_amplification.html

# (Python 3 setup cell omitted)
# https://developers.transcriptic.com/docs/how-to-write-a-new-protocol
# https://secure.transcriptic.com/_commercial/resources?q=water

import json
import autoprotocol
from autoprotocol.protocol import Protocol
p = Protocol()

# 3 cols: 0=template+primers+mastermix, 1=primers+mastermix, 2=water
# 3 rows: A, B, C repeated
experiment_name = "puc19_m13_v1"
```

```
16
17 inv = {}
18 inv['SensiFAST SYBR No-ROX'] = "rs17knkh7526ha"
19 inv['water'] = "rs17gmh5wafm5p"
20 inv['M13 Forward (-20)'] = "rs17tcpupe7fdh"
21 inv['M13 Reverse (-48)'] = "rs17tcph6e2qzh"
22 inv['pUC19'] = "rs17tcqmncjfsh"
23
24 #-----------------------------------------------------------
25 # Provisioning things for my PCR
26 #
27 # Provision a 96 well PCR plate (https://developers.transcriptic.com/v1.0/docs/containe
28 # Type      Max       Dead    Safe    Capabilities
29 # 96-pcr    160 µL    3 µL    5 µL    pipette, sangerseq, spin, thermocycle, incubate, g
30 #
31 pcr_plate = p.ref("pcr_plate", cont_type="96-pcr", storage="cold_4")
32
33 #-----------------------------------------------------------
34 # SYBR-including mastermix
35 # http://www.bioline.com/us/downloads/dl/file/id/2754/sensifast_sybr_no_rox_kit_manual.p
36 # Instructions: 10ul mastermix + 0.8ul primer (400nM) + 0.8ul primer (400nM) + ⩽8.4ul t
37 #
38
39 #mastermix_tube = p.ref("mastermix_tube", cont_type="micro-2.0", storage="cold_20")
40 for well in ["A1", "B1", "C1", "A2", "B2", "C2"]:
41     p.provision(inv['SensiFAST SYBR No-ROX'], pcr_plate.wells(well), "10:microliter")
42
43
44 #-----------------------------------------------------------
45 # M13 primers
46 # I choose m13 (-20) and (-48) because of similar Tm. This amplifies ~110bp including pr
47 #
48 # 100pmol == 1ul, since Transcriptic dilutes the 1300-1900pmol into 13-19ul (depending o
49 # I want 400nM in the final 20ul according to the SensiFAST documentation (=8pmol in 20
50 # 1ul primer in 12ul total equals 8pmol/ul
51 #
52
53 # http://www.idtdna.com/pages/products/dna-rna/readymade-products/readymade-primers
54 # Name                sequence                          Tm    Anhyd.  pmoles in 10ug
55 # M13 Forward (-20)   GTA AAA CGA CGG CCA GT             53.0  5228.5  1912.6
56 # M13 Forward (-41)   CGC CAG GGT TTT CCC AGT CAC GAC    65.5  7289.8  1371.7
57 # M13 Reverse (-27)   CAG GAA ACA GCT ATG AC             47.3  5212.5  1918.3
58 # M13 Reverse (-48)   AGC GGA TAA CAA TTT CAC ACA GG     57.2  7065.7  1415.2
59
60 primer_tube = p.ref("primer_tube", cont_type="micro-2.0", storage="cold_20")
61 p.provision(inv['M13 Forward (-20)'], primer_tube.wells(0), "1:microliter") # fwd -20
62 p.provision(inv['M13 Reverse (-48)'], primer_tube.wells(0), "1:microliter") # rev -48
63 p.provision(inv['water'], primer_tube.wells(0), "10:microliter") # water
64
65
66 #-----------------------------------------------------------
67 # pUC19
68 # 1000ug/ml → 1ul = 1ug == 1000ng. Add 1ul to 49ul to get ~20ng/ul
69 # Then I can transfer 5ul to get 100ng total
70 #
71 template_tube = p.ref("template_tube", cont_type="micro-2.0", storage="cold_20")
72 p.provision(inv['pUC19'], template_tube.wells(0), "1:microliter")
73 p.provision(inv['water'], template_tube.wells(0), "49:microliter") # water
74
75
76 #-----------------------------------------------------------
```

```
# Move all the reagents into the pcr plate
# The "dispense" command does not work because it needs ⩾10ul per dispense
# in increments of 5ul
#
for wells, ul in (["A1","B1","C1"], 4), (["A2", "B2", "C2"], 9), (["A3", "B3", "C3"], 20
    for well in wells:
        p.provision(inv['water'], pcr_plate.wells(well), "{}:microliter".format(ul))

p.transfer(template_tube.wells(0), pcr_plate.wells(["A1", "B1", "C1"]), "5:microliter")
p.transfer(primer_tube.wells(0), pcr_plate.wells(["A1", "B1", "C1", "A2", "B2", "C2"]),

#-----------------------------------------------------------
# Thermocycle, with a hot start (95C for 2m)
# Based on http://www.bioline.com/us/downloads/dl/file/id/2754/sensifast_sybr_no_rox_kit
# I also found http://www.environmental-microbiology.de/pdf_files/M13PCR_13jan2014.pdf
# p.seal before thermocycling is enforced by transcriptic
#
p.seal(pcr_plate)
p.thermocycle(pcr_plate, [{
      "cycles": 1, "steps": [{
          "temperature": "95:celsius",
          "duration": "2:minute"}]
    }, {
      "cycles": 40, "steps": [{
          "temperature": "95:celsius",
          "duration": "5:second"}, {
          "temperature": "60:celsius",
          "duration": "20:second"}, {
          "temperature": "72:celsius",
          "duration": "15:second", "read": True}]
    }],
    volume="20:microliter", # volume is optional
    dataref="qpcr_{}".format(experiment_name),
    # Dyes to use for qPCR must be specified (tells transcriptic what aborbance to use?)
    dyes={"SYBR": ["A1", "B1", "C1", "A2", "B2", "C2", "A3", "B3", "C3"]},
    # standard melting curve parameters
    melting_start="65:celsius",
    melting_end="95:celsius",
    melting_increment="0.5:celsius",
    melting_rate="5:second")

#-----------------------------------------------------------
# Run a gel
# agarose(8,0.8%): 8 lanes, 0.8% agarose 10 minutes recommended
# 10 microliters is used in the example documentation
# ladder1: References at 100bp, 250bp, 500bp, 1000bp, and 2000bp.
# The gel already includes SYBR green
#
p.gel_separate(pcr_plate.wells(["A1", "B1", "C1", "A2", "B2", "C2", "A3", "B3"]),
               "10:microliter", "agarose(8,0.8%)",
               "ladder1", "10:minute", "gel_{}".format(experiment_name))


#-----------------------------------------------------------
# Analyze and output the protocol
#
jprotocol = json.dumps(p.as_dict(), indent=2)
print(jprotocol)
open("protocol.json",'w').write(jprotocol)
uprint("Analyze protocol")
!echo '{jprotocol}' | transcriptic analyze
```

Antha

Code Walkthrough - Antha OS Documentation

Every element in the Antha environment needs a unique element name. For this element, the name is "Aliquot".

The Aliquot element takes nine pieces of data as its Parameters. **Parameters are the non-physical inputs for an element**, like temperature, duration, or volume. The physical inputs for an element are described in the Inputs block.

The **Data block of an Antha element defines the information produced by the element as a data output.** These include things like final sample volume, number of aliquots performed, or thaw-time required.

For this Aliquot element, there is a single data output WellsUsed. The WellsUsed data output is as the name suggests a list of wells in a specific plate that have been used by this element. The map structure is used here to associate the list of well coordinates that will contain an aliquot to the specified output plate name.

The **Inputs block of an element file defines the physical materials required by the element.** These include things like solution sample, DNA part, or multi-well plate.

The Outputs section lists the physical things that are generated by an element.

The Setup block is performed the first time that an element is executed. This can be used to perform any configuration that is needed globally for the element, and is also used to define any special setup that may be needed for groups of concurrent tasks that might be executed at the same time. Any variables that need to be accessed by the Steps function globally can be defined here as well, but need to be handled with care to avoid concurrency problems.

At this current time, the Setup block is not supported by Antha, but will be in future releases.

The heart of an Antha element is **the Steps block, which defines the actual steps taken to transform a set of Parameters and Inputs into Data and Outputs.** The Steps block is a kernel function, meaning it shares no information for every concurrent sample that is processed, and defines the workflow to transform a single block of inputs and samples into a single set of outputs, even if the element is operating on an entire array (such as micro-titre plate of samples at once).

```
1  // https://github.com/antha-lang/elements/tree/master/an/AnthaAcademy/Lesson1_Commands/I
2
3  // Example protocol demonstrating the use of the Sample function.
4  // The Sample function is not sufficient to generate liquid handling instructions alone,
5  // We would need a Mix command to instruct where to put the sample
6  // We can either modify the code to add this or wire the output Sample into the Lesson1E
7  // Any comment placed here directly above the protocol name will appear in AnthaOS as th
8  //
9  // Concepts covered:
10 // Anatomy of an Antha element
11 // types
12 // Volume
13 // Comments and AnthaOS
14 // LHComponent
15 // Sampling
```

```
// Reading Code
// imports
// functions

protocol Lesson1A_Sample // this is the name of the protocol Lessonthat will be called i

// the mixer package must be imported to use the Sample function
import (
  "github.com/antha-lang/antha/antha/anthalib/mixer"
)

// Input parameters for this protocol (data)
Parameters {
  // antha, like golang is a strongly typed language in which the type of a variable mus
  // In this case we're creating a variable called SampleVolume which is of type Volume;
  // the type system allows the antha compiler to catch many types of common errors befo
  // the antha type system extends this to biological types such as volumes here.
  // functions require inputs of particular types to be adhered to.
  // Any text written above any of the parameters, Data, Inputs and Outputs variables
  // will appear in AnthaOS as annotations.
  SampleVolume Volume
}

// Data which is returned from this protocol, and data types
Data {
  // Antha inherits all standard primitives valid in golang;
  //for example the string type shown here used to return a textual message
  Status string
}

// Physical Inputs to this protocol with types
Inputs {
  // the LHComponent is the principal liquidhandling type in antha
  // the * signifies that this is a pointer to the component rather than the component i
  // most key antha functions such as Sample and Mix use *LHComponent rather than LHComp
  Solution *LHComponent
}

// Physical outputs from this protocol with types
Outputs {
  // An output LHComponent variable is created called Sample
  Sample *LHComponent
}

Requirements {

}

// Conditions to run on startup
Setup {

}

// The core process for this protocol, with the steps to be performed
// for every input
Steps {

  // Programming is typically made up of a series of functions.
  // Functions, like mathematical functions and like the antha elements themselves,
  // are black boxes which process some input arguments to produce outputs.

```

```
  // In this line of code we have a variable on the left called Sample.
  // We initialised this variable as an LHComponent above in the Outputs section.
  // Because the Sample is to the left of an = sign,
  // the value of Sample will be updated as the product of the mixer.Sample function to
  // At the top of the element file we can see that we import a library which ends with
  // Here we are using a function called Sample from the mixer library.
  // This demonstrates one use of a full stop when reading code: accessing code stored i
  // In Antha, as with Golang, any code which is imported from a package will always sta
  // We can tell Sample is a function here since it is proceded by parenthesis( ).
  // The contents of the parentheses, Solution and SampleVolume, are the input
  // arguments to the function. We can find out what a specific function requires as inp
  // In the mixer library the function signature can be found,
  // here it is:
  // func Sample(l *LHComponent, v Volume) *LHComponent {
  // The function signature  shows that the function requires a *LHComponent and a Volum
  Sample = mixer.Sample(Solution, SampleVolume)

  // The Sample function is not sufficient to generate liquid handling instructions alon
  // We would need a Mix command to instruct where to put the sample
  // We can either modify the code to add this or wire the output Sample into the Lesson

  // we can also export data only outputs.
  // In this case we'll use quotations to write a message as a string like this:
  Status = "Lesson 1A_Sample has been a success, now wire the corresponding output into

}

// Run after controls and a steps block are completed to
// post process any data and provide downstream results
Analysis {
}

// A block of tests to perform to validate that the sample was processed
//correctly. Optionally, destructive tests can be performed to validate
//results on a dipstick basis
Validation {

}
```