Structured identifiers in TavernaProv and wfdesc

The identifier rules include important aspects such as

- Design identifiers for use by others
- Avoid embedding meaning
- Opt for simple durable web resolution

When we mint identifiers as part of a Taverna workflow run (which might be running on a desktop computer behind a firewall), we mint structured URIs using a combination of UUIDs and structural information.

example:

http://ns.taverna.org.uk/2011/run/385c794c-ba11-4007-a5b5-502ba8d14263/

identifies the Taverna workflow "run" 385c794c-ba11-4007-a5b5-502ba8d14263a - each run gets a new UUID.

One advantage here is that our server, while not having access to the distributed workflow runs across the world, can still at least say that it is a workflow run - so it redirects to:

http://guess.taverna.org.uk/run/385c794c-ba11-4007-a5b5-502ba8d14263/

Each run generates workflow data items with internal identifiers, e.g.,

http://ns.taverna.org.uk/2011/data/385c794c-ba11-4007-a5b5-502ba8d14263/list/c2f58d3e-8686-40a5-b1cd-b797cd18fbb7/false/1

Here you would think c2f58d3e-8686-40a5-b1cd-b797cd18fbb7 would be sufficient to identify this data item, but we choose to include the run UUID as well, as data is always generated as part of a workflow run and this can help find the data.

Because of "list/" we can also say that it's a collection (it will always be a collection) - meaning that the guesser service can say it's a prov:Collection (but it does not say what elements are part of this collection).

For the data, we deliberately did NOT include the UUID of the workflow DEFINITION that was run, as that could be leaking information, e.g., that you are running someone else's workflow.

Similarly elements of a collection have their own UUID-based identifiers, as they could be part of multiple collections.

But when the hierarchy is fixed, and the identifier only makes sense within a hierarchical structure, then I don't mind it being present in the identifier.

Hierarchical structures go hand-in-hand with typing, e.g., the slightly lengthy identifier:

http://ns.taverna.org.uk/2010/workflowBundle/01348671-5aaa-4cc2-84cc-477329b70b0d/workflow/Hello_Anyone/processor/Concatenate_two_strings/in/string1

It's "OK" that the URL is not very friendly, as it would only appear within a detailed provenance trace - but it's important that the identifier is unique and consistent wherever the particular workflow definition is run - that way we can relate provenance across workflow runs.

Here you can see multiple global and local identifiers and associated types:

type workflow bundle: 01348671-5aaa-4cc2-84cc-477329b70b0d (a UUID)
  type workflow: Hello_Anyone (a potentially nested workflow in the bundle)
    type processor: Concatenate_two_strings (a step in the workflow)
      type input port: string1 (a particular input parameter of that step)

This being a HTTP URI, the the 'guesser' service can recreate the above structure, even without knowing anything else about the workflow definition.

The local identifier 'string1' depends on each of the above to be contextualized and globally unique.

While we could just have minted UUIDs for each part of the workflow, e.g.,
http://ns.taverna.org.uk/2010/822fd076-5f85-4eee-809e-0403e04d4f55
this actually adds new problem - now the identifiers all look the same, and are useless for manual debugging (e.g., just having a "P" prefix for protein type helps humans wade through debug outputs).

Another problem comes in versioning - when does our workflow processor step input port "change" and get a new identifier? What if the entire rest of the workflow evolves? Do we re-assign every UUID across the structure, or can the same UUID appear in many workflow (versions)? Each approach has advantages and disadvantages.

By making the unit of change the workflow bundle (that is, the ZIP file of the workflow definition you can download), we also make a single point to update the hierarchical identifiers for all the constituent parts for any change. This means, code-wise, we have to be careful to use relative identifiers internally, and absolute identifiers externally.

BTW, we keep a provenance trace of those UUID ancestors.