




Design and implementation of the preservation component catalogue

Authors

Donal Fellows (University of Manchester), Markus Plangg (University of Technology Berlin)

May 2014

This work was partially supported by the SCAPE Project. The SCAPE project is co-funded by the European Union under FP7 ICT-2009.4.1 (Grant Agreement number 270137).

This work is licensed under a CC-BY-SA International License 

Executive Summary

This deliverable consists of the updated myExperiment service, the published SCAPE ontology, and two supporting tools. The update to myExperiment makes it able to work as a component repository, the SCAPE ontology defines the semantic annotation terms that are used in the component profiles on myExperiment, and the tools make it easier to determine the “correctness” of a component and to create and edit a component profile.

The Component Repository is at:

<http://www.myexperiment.org/>

The SCAPE Ontology is at:

<http://purl.org/DP/components/1.0>

The SCAPE Component Profiles are (in un-deployed form) at:

<https://github.com/openplanets/scape-component-profiles>

The supporting tools are at:

<https://github.com/myGrid/component-validator>

<https://github.com/myGrid/component-profile-creator>

Table of Contents

Executive Summary	iii
1 Introduction	1
1.1 Taverna Components Abstract Model	1
2 Component Registry	2
2.1 User Interface.....	2
2.2 Configuration of myExperiment to Support SCAPE	4
2.3 Application Programming Interface.....	4
2.3.1 API for Components.....	5
2.3.2 API for Component Families	7
2.3.3 API for Component Profiles	9
3 SCAPE Component Ontology.....	10
3.1 Workflow-level annotations.....	11
3.2 Port-level annotations.....	11
3.3 Internal annotations.....	11
3.4 The SCAPE component profiles.....	12
4 Supporting Tools.....	12
4.1 Component Validator Library.....	12
4.2 Component Profile Editor	14
5 Glossary	15

1 Introduction

The SCAPE project uses Taverna as its workflow system of choice. Workflows in Taverna can be split up into pieces — sub-workflows — that can be shared via the myExperiment workflow repository service. However, these sub-workflows do not hide any information about their nature; they remain relatively difficult to use. They were also comparatively difficult to search for, and the API for doing so was difficult for any tool other than Taverna to use, especially where that tool was not just a simple user interface.

To resolve this, we have designed the *Taverna Component* concept, which is a semantically annotated sub-workflow that can be placed in a containing workflow without exposing the details of its implementation. To ease discovery of components, they are grouped into *families* of related components that share a common principle (e.g. migration actions suitable for application to a web archive). Each family encodes part of its nature as a *profile*, which is a document that describes the interface of the component, what annotations (including semantic annotations drawn from any ontology that it nominates) it may have, and how to transform any errors generated within the component into syntax that the users of the component can understand.

However, merely having the component concept is insufficient. We also require that components be discoverable and shareable *in reality*. We therefore have extended the myExperiment workflow repository with the capability to be a *component repository* as well. This allows users to find components by three mechanisms: general search within myExperiment, exploration of the component service panel in the Taverna Workbench, and semantic search via the repository API. It also leverages the existing sharing platform of myExperiment, and allows for the presentation of additional information not normally required (e.g. the assessment of the validity of a component against its profile).

We also define an ontology specifically for the components used in the SCAPE project, together with tools for the validation of components against their profile, and for creating and updating a profile.

1.1 Taverna Components Abstract Model

Components in Taverna follow an abstract model as set out in Figure 1 (blue blocks, and the registry in green, are part of the component interface; red blocks are realizations in Taverna, and orange blocks relate to semantic annotations).

Components are versioned entities that can be used in a Taverna Workflow. These components are aggregated into component families for the purposes of discovery and organization. Each component family also references a component profile that its member components should conform to. The profile defines what annotations should be present, what ontologies may supply those annotations, how many input and output ports there should be, etc. The components, component families and component profiles are all stored in a component repository; that component repository is responsible for providing operations over these entities (e.g. lookup, semantic search).

A version of a component is realized by a Taverna Workflow, with including a component in a workflow being conceptually similar to placing a sub-workflow in that consuming workflow. The annotations are placed on the workflow that defines the component, on the ports of that workflow, and possibly also on the activities (processing nodes) in the workflow.

The publicly accessible realization of the component repository concept is myExperiment.

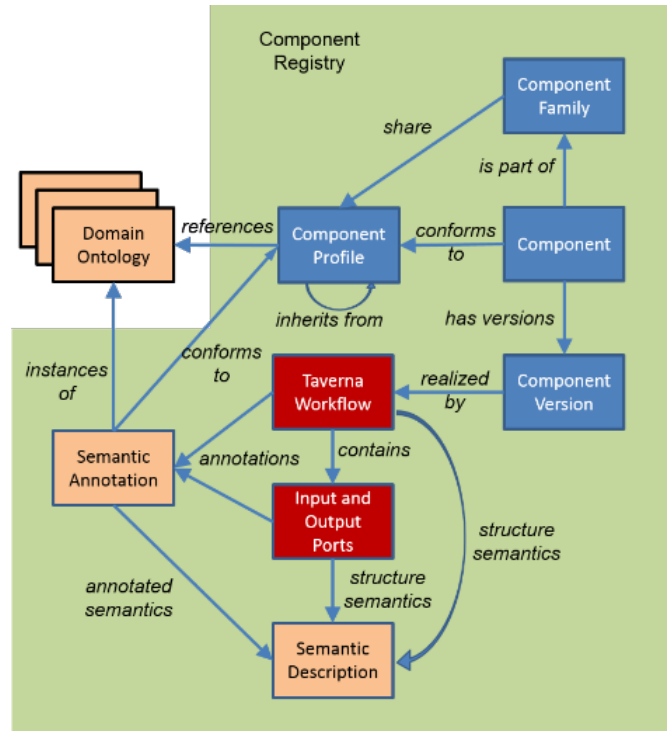


Figure 1: Abstract Model of Taverna Components Ecosystem

2 Component Registry

The SCAPE project uses myExperiment (<http://www.myexperiment.org/>) as its component repository as it provided a substantial fraction of the desired functionality with only minimal changes, allowing a more professional component repository to be delivered. However, becoming a component repository as well as a workflow repository has entailed some alterations to myExperiment. This section highlights the key ones.

Note that all the changes described here are present in our production deployment, and are in use in projects¹ outside SCAPE. However, the SCAPE project is the most extensive known user of semantic annotations in components; the other users currently tend to leave their semantics implicit.

2.1 User Interface

We have added SCAPE project styling (see Figure 2) to items (workflows, components, files, packs, etc.) that are shared with the SCAPE user group on myExperiment. This styling is not enabled by default, but can be added via the user's management view for the item.

¹ Notably the BioVeL project, grant agreement #283359, <https://www.biovel.eu/>. They have focused much more on the hardening of the GUI for components in the Taverna Workbench, testing of the generation of provenance out of components, plus a feature for remapping of exceptions from inferior processors (that feature is unused in SCAPE).

The user interface to myExperiment has had relatively few changes to adapt to components, as they fit relatively closely with the existing models of workflows, packs and files (which components, families and profiles can be regarded as being specializations of). It should be noted that while it is possible to create components that are in the repository but not shared with other users, this is *not recommended* at all because it completely prevents reuse.

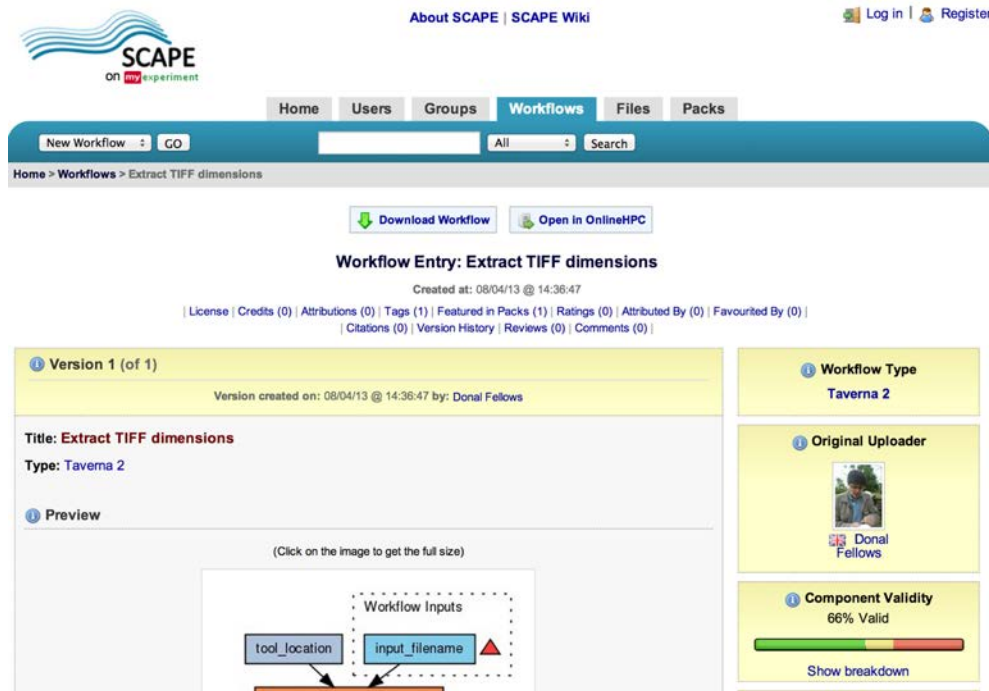


Figure 2: myExperiment Component View from <http://www.myexperiment.org/workflows/3498.html>

The main interface change (apart from now declaring component profiles to be such) is that we now include an extra side panel for components, the "Component Validity", which shows how closely the component conforms to its profile (formally, to the profile of the primary component family that contains the component). It uses a "traffic light" bar to indicate how well the profile is satisfied, where green indicates total satisfaction of a condition, red indicates total failure to satisfy a condition, and yellow indicates *technical* satisfaction, but in a way that the user is highly unlikely to expect (i.e., the validation issues a warning, for example, when a mentioned semantic annotation is required to be present on all ports but there are no ports at all). The summary figure above it indicates the total of the full satisfaction and warning states as a proportion of the total.

Clicking on the "Show breakdown" link causes the side panel to expand to include explanatory text showing why the assessment is what it is (see Figure 3).

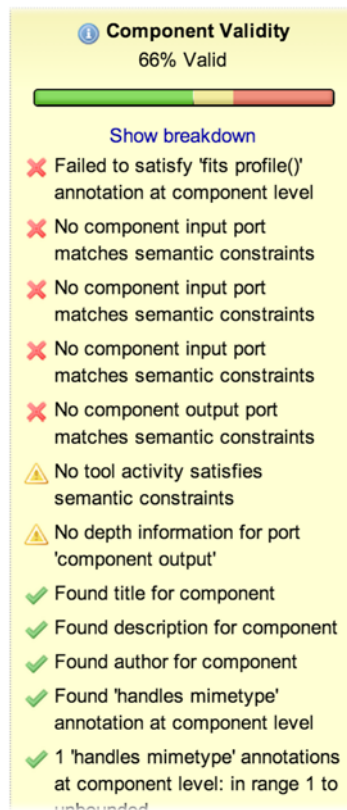


Figure 3: The Expanded Component Validity Box from <http://www.myexperiment.org/workflows/3498.html>

By policy, failure to conform to a profile does *not* prevent the use of the component; it formally merely constitutes the opinion of the myExperiment service on how well the profile is satisfied.

Although there is a semantic search capability in the API (see Section 2.3.1), we do not surface this to the user interface as our user testing indicated that the complexity of writing search terms exceeded what we could expect a reasonable user to use.

2.2 Configuration of myExperiment to Support SCAPE

We have performed a number of adaptations to myExperiment to support SCAPE. In particular, we have a user group for the project ("SCAPE", <http://www.myexperiment.org/groups/490.html>) and we support the branding of an artefact with the SCAPE logo and colour palette (provided the item is shared with the SCAPE group). The branding can be seen in Figure 2.

We have a number of component families and component profiles provided for the SCAPE project. We provide a component profile for each of the major tasks performed by SCAPE components (Content Migration, Characterisation and Quality Assurance) and component families that use those profiles in each of the thematic areas (Audio/Video, Documents, Images, Scientific Data and Webpages). These component families are also the set that are exposed by default in the Taverna Workbench 2.5 for Digital Preservation. The components within these families are defined and provided by the Preservation Components subproject workpackages.

2.3 Application Programming Interface

We have also extended the REST interface to myExperiment with additional operations to make working with components from programs easier. The majority of this API is used within the Taverna

Workbench to access and manipulate components within the repository, and it is substantially more efficient than the older workflow and pack APIs that it replaces.

This new API is also documented on the myExperiment Developer site at <http://wiki.myexperiment.org/index.php/Developer:Components>; all main URLs are with respect to the root URL of the repository.

Note that all the APIs below can be used either with or without authentication. Where a query API (GET) is used without authentication, only public information is returned. Where a modification API (PUT, POST, DELETE) is used without authentication, it will always be rejected. Authentication is done via HTTP Basic Authentication headers.

2.3.1 API for Components

It can be seen that the URLs of components contain an ID; this ID is shared between components and workflows, as components are subclasses of workflows.

Discovery of components

GET /components.xml

Fetches the list of all components that the user may see (i.e., publicly viewable components, components that they own, and those that they have been granted elevated access to).

Example of response (truncated):

```
<workflows>
  <workflow id="3417" version="2"
    resource="http://www.myexperiment.org/workflows/3417"
    uri="http://www.myexperiment.org/workflow.xml?id=3417">
    Migration Imagemagick convert no compression
  </workflow>
  <workflow id="3364" version="1"
    resource="http://www.myexperiment.org/workflows/3364"
    uri="http://www.myexperiment.org/workflow.xml?id=3364">
    Migration ffmpeg audio to wav pcm_s32le
  </workflow>
</workflows>
```

GET /components.xml?component-family={uri}

Fetches the list of components that are a member of the given family and that the user may see. The {uri} parameter must be a fully qualified *pack* URI of a family in the same repository, for example <http://www.myexperiment.org/packs/123>. Note that this effectively acts as a filter on the results of fetching all components.

GET /components.xml?query={qry}&prefixes={pfx}

Fetches the list of components that satisfy the given SPARQL² query (the {qry} parameter). The {pfx} parameter is used to specify the URI prefixes used in the query. The repository processes the query by using the parameters in the template below, and then evaluating the resulting SPARQL query against a read-only view of an RDF model of the space of workflows.

² Pérez, Jorge, Marcelo Arenas, and Claudio Gutierrez. "Semantics and Complexity of SPARQL." *The Semantic Web-ISWC 2006*. Springer Berlin Heidelberg, 2006. 30-43.

The repository filters the resulting list of workflow URIs by what the current user has permission to see.

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX wfdesc:<http://purl.org/wf4ever/wfdesc#>
[SPARQL {pfx} parameter here]

SELECT DISTINCT ?workflow_uri WHERE {
  GRAPH ?workflow_uri {
    ?w a wfdesc:Workflow .
    [SPARQL {qry} parameter here]
  }
}
```

GET /components.xml?query={qry}&prefixes={pfx}&component-family={uri}
Performs a SPARQL search (as above) and filters the result for membership of the given component family. This is the composition of the above two query styles.

Fetching the metadata for a component

GET /component.xml?id={id}&elements={elems}
Fetches the description of the component with ID {id}. The optional {elems} parameter is a comma-separated list of what fields to fetch; the most typically useful fields are title, description, content-uri, versions and component-families (note that this field is a list result since a component may logically belong to multiple families, even if this is rare in practice).

Example response (fetching the above named fields, with some line-breaks):

```
<workflow id="3498" version="1"
  uri="http://www.myexperiment.org/workflow.xml?id=3498"
  resource="http://www.myexperiment.org/workflows/3498">
  <title>Extract TIFF dimensions</title>
  <description>
    Component that extracts the dimensions of a TIFF image.
  </description>
  <content-uri>
    http://www.myexperiment.org/workflows/3498/download/
    extract_tiff_dimensions._32796.t2flow
  </content-uri>
  <versions>
    <workflow id="4739" version="1"
      uri="http://www.myexperiment.org/workflow.xml?id=3498&version=1"
      resource="http://www.myexperiment.org/workflows/3498?version=1">
        Extract TIFF dimensions
      </workflow>
    </versions>
  <component-families>
    <component-family>
      http://www.myexperiment.org/packs/416
    </component-family>
  </component-families>
</workflow>
```

The content of the `content-uri` field is the exact URL from which the component definition (a `.t2flow` document) can be downloaded from.

Creation of a component

POST `/component.xml?elements={elems}`

To create a component, send a suitable document to this API call. In particular, you need to specify the title, the description, the component family (using the `.../packs/{id}` URI), the content-type and the content (i.e., serialized implementing Taverna workflow) itself. The optional `{elements}` parameter describes a comma-separated list of what elements should be in the response (just as when retrieving a component); it is recommended that this be `title,description`.

An example of a creation request (without the main payload) is:

```
<?xml version="1.0"?>
<workflow>
  <title> Example Component </title>
  <description> This is an example of a component. </description>
  <component-family>
    http://www.myexperiment.org/packs/592
  </component-family>
  <content-type>application/vnd.taverna.t2flow+xml</content-type>
  <content encoding="base64" type="binary">
    [base64-encoded .t2flow file here]
  </content>
</workflow>
```

The response to a successful creation operation is a document like this:

```
<?xml version="1.0"?>
<workflow id="12345" version="1"
  resource="http://www.myexperiment.org/workflows/12345"
  uri="http://www.myexperiment.org/workflow.xml?id=12345">
  <title> Example Component </title>
  <description> This is an example of a component. </description>
</workflow>
```

Addition of a new component version

POST `/component.xml?id={id}&elements={elems}`

Clients create a new version by posting the identical document to when creating a new component, except that the target URL for the POST is now the component URL for a specific component. The request and response documents are identical to above.

Deletion of a component

DELETE `/workflow.xml?id={id}`

Components are deleted by using a simple DELETE message, supplying the component's ID within the repository. Because this is really an operation on the superclass type, the message should be sent to the `/workflow.xml` endpoint, not the `/component.xml` endpoint.

2.3.2 API for Component Families

Component families are a subclass of pack (collection of related workflows and files) in the myExperiment API; any ID for a component family is also inherently an ID of a pack.

Discovery of component families

GET /component-families.xml

Lists all component families that the current user may see. The result list is in the form:

```
<component-families>
  <pack id="314" version=""
    resource="http://www.myexperiment.org/packs/314"
    uri="http://www.myexperiment.org/pack.xml?id=314">
    Task Data
  </pack>
  <pack id="414" version=""
    resource="http://www.myexperiment.org/packs/414"
    uri="http://www.myexperiment.org/pack.xml?id=414">
    SCAPE Image Migration Action
  </pack>
</component-families>
```

GET /component-families.xml?component-profile={uri}

Lists all component families that the current user may see and which have members that should conform to the given profile (given by the resource URI for a particular profile).

Creation of a component family

POST /component-family.xml&elements={elems}

This creates a component family, with the optional {elems} indicating which elements are desired in the response record. The recommended elements are title, description.

The request document, which *must* include a component-profile element that refers to the *file* resource for the profile, is formatted like this:

```
<pack>
  <title> The family title </title>
  <description> A description of the component family. </description>
  <component-profile>
    http://www.myexperiment.org/files/12345
  </component-profile>
</pack>
```

The format of the response document (with recommended parameters) is:

```
<pack id="23415" version=""
  resource="http://www.myexperiment.org/packs/23415"
  uri="http://www.myexperiment.org/pack.xml?id=23415">
  <title> The family title </title>
  <description> A description of the component family. </description>
</pack>
```

Deletion of a component family

DELETE /component-family.xml?id={id}

This deletes the given family, but only if *all* the components within the family have also been deleted.

2.3.3 API for Component Profiles

Component profiles are kinds of file objects in the myExperiment model; they share IDs.

Discovery of component profiles

GET /component-profiles.xml

This lists all component profiles that the current user has permission to read. An example of the response is:

```
<component-profiles>
  <file id="904" version="6"
    resource="http://www.myexperiment.org/files/904"
    uri="http://www.myexperiment.org/file.xml?id=904">
    Migration Action Component
  </file>
  <file id="905" version="4"
    resource="http://www.myexperiment.org/files/905"
    uri="http://www.myexperiment.org/file.xml?id=905">
    Characterisation Component
  </file>
</component-profiles>
```

Description of a particular component profile

GET /file.xml?id={id}&elements={elems}

This API call retrieves detailed information about a particular profile (identified by its ID). The optional {elems} parameter is a comma-separated list of fields to retrieve; the *recommended* set is title,description,content-uri, with the content-uri indicating where to retrieve the content of the profile document.

Example response (with content-uri split over multiple lines):

```
<file id="904" version="6"
  uri="http://www.myexperiment.org/file.xml?id=904"
  resource="http://www.myexperiment.org/files/904">
  <title>Migration Action Component</title>
  <description>
    A SCAPE component for migration actions
  </description>
  <content-uri>
    http://www.myexperiment.org/files/904/download/
    MigrationActionComponentProfile.xml
  </content-uri>
</file>
```

Creation of a component profile

POST /component-profile.xml?elements={elems}

This creates a component profile. The profile *must* have the content-type field set to application/vnd.taverna.component-profile+xml or the operation will fail to create a profile. The {elems} parameter may be supplied to select what elements should be in the response; the recommended value is title,description.

Example creation request:

```
<file>
  <title>Profile Title</title>
  <description>A description of the profile.</description>
  <filename>profile.xml</filename>
  <content-type>
    application/vnd.taverna.component-profile+xml
  </content-type>
  <content encoding="base64" type="binary">
    [base-64 encoded profile contents]
  </content>
</file>
```

The response will be a file element with `id`, `resource` and `uri` attributes filled in.

```
<file id="12345" version=""
  resource="http://www.myexperiment.org/files/12345"
  uri="http://www.myexperiment.org/file.xml?id=12345">
  <title> Profile title </title>
  <description> A description of the profile. </description>
</file>
```

Deletion of a component profile

```
DELETE /component-profile.xml?id={id}
```

This deletes the component profile with ID equal to `{id}`, provided it is unused in any component family.

3 SCAPE Component Ontology

To support the creation of machine-comprehensible components, the SCAPE project publishes an ontology of terms. The formal address of this ontology, <http://purl.org/DP/components/1.0>, is a PURL (persistent URL) that redirects to the real publication location (currently a hosted system at Technical University of Vienna) as that allows for flexibility in where the data is located without any requirement to modify existing profiles or components.

The ontology (see Figure 4, Figure 5 and Figure 6) is designed to complement workflows by providing simple OWL statements about workflows and workflow parts. The statements are added as semantic annotations to the relevant parts. Depending on the component's purpose, the component profile prescribes mandatory and optional annotations.

The annotations can be specific individuals already defined in the ontology, e.g. `#MigrationAction`, to identify the profile of a workflow. Since the ontology is pre-populated with these individuals, they can be easily selected when creating a component. Additionally this provides a unified vocabulary to identify workflow parts and their purpose.

For more complex and dynamic data, only the required OWL class of the annotation is defined in the profile. This allows the workflow creator to add anonymous individuals containing user-defined data and additional properties, e.g. `#MigrationPath`.

The ontology classes, properties and individuals we have defined can be roughly grouped by the workflow part they annotate.

3.1 Workflow-level annotations

Annotations for the workflow itself (see Figure 4) contain the profile they adhere to. This allows simpler querying by component type. To allow workflow discovery, the component's supported data formats must be specified as MIME-types. To avoid repetitive data, the MIME-type's subtype name can be replaced with a * wildcard indicating that all subtypes are supported.

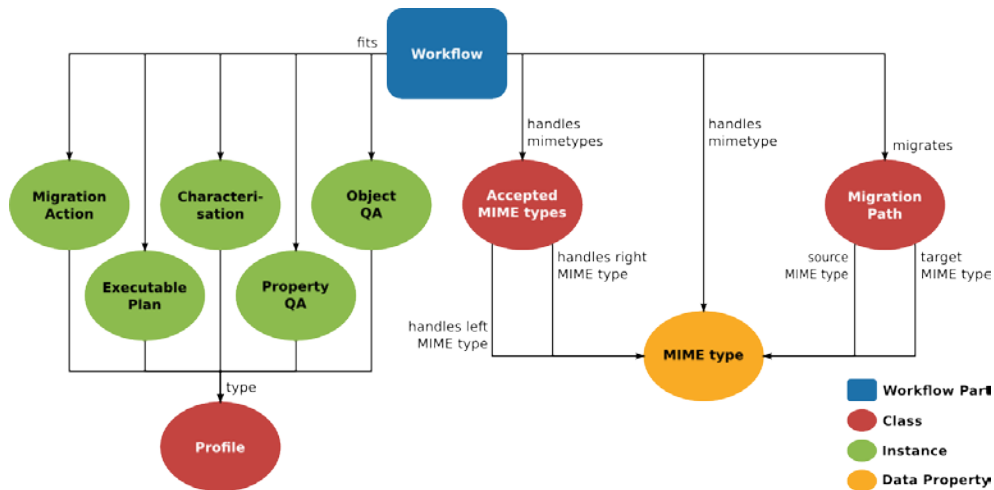


Figure 4: Ontology of Whole-Component-Level Annotations

3.2 Port-level annotations

To support automated execution and composition of components the inputs and outputs must be identified (see Figure 5). For input ports the ontology provides the #accepts property to distinguish between input objects, parameters and other data. Parameter ports can further be annotated with predefined values to simplify execution for the end user. Output ports should use the #provides property to identify the data they produce, e.g. migrated files, or measures from the measure catalogue³. The ontology optionally allows the component designer to restrict the legal data-types of inputs and outputs.

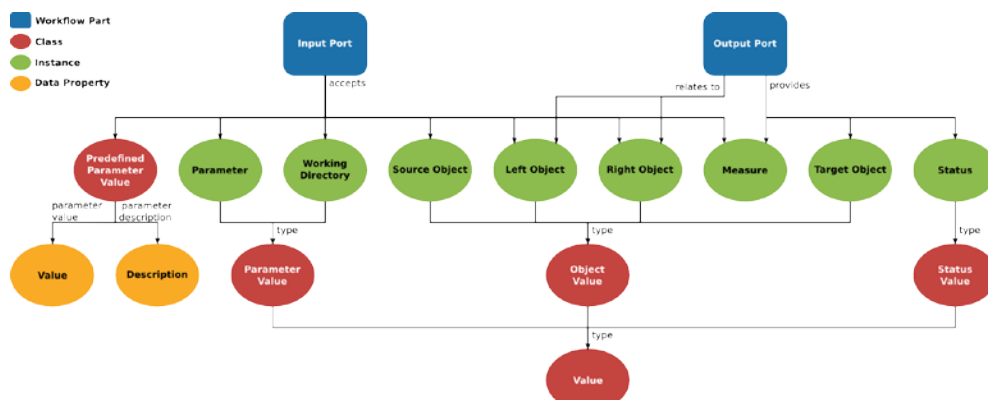


Figure 5: Ontology of Port-Level Annotations

3.3 Internal annotations

In case the component uses external command line tools, the ontology (see Figure 6) allows the component creator to specify the tools installation environment with the

³ <http://purl.org/DP/quality/measures>

#requiresInstallation property. The installation is bound to an environment, with some common environments pre-defined as individuals in the ontology. The execution is often dependent on the specific version of the external tool. Using the #dependsOn property allows defining the dependency. This additionally allows discovery by the used tool and its license. Support for specifying Debian packages, RPMs and Maven repositories as installation sources is provided by the #hasSourceConfiguration property and its subtypes.

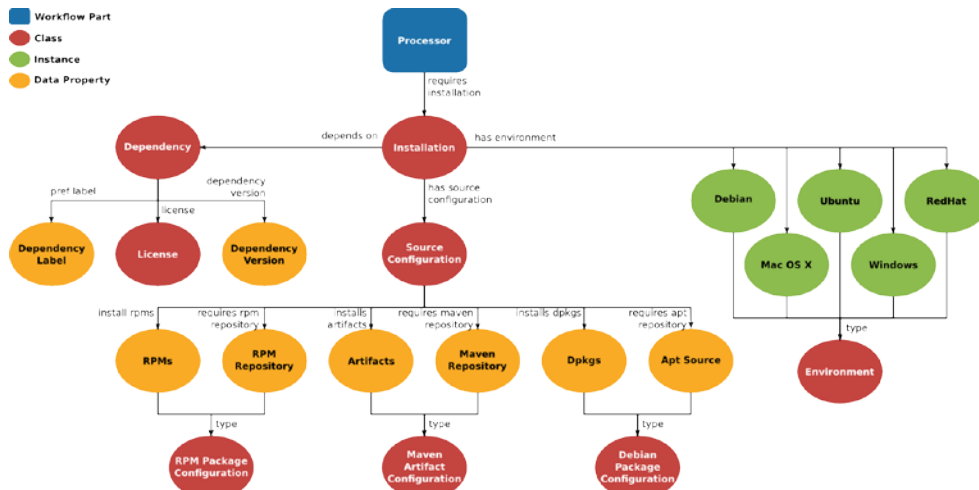


Figure 6: Ontology of Tool-Level Annotations

3.4 The SCAPE component profiles

These ontologies are used to construct component profiles (which we have developed on GitHub, <https://github.com/openplanets/scape-component-profiles>) and these are then uploaded to myExperiment to deploy them into general use. These profiles, described in more depth at the above page, formalize the constraints that ensure interoperability between the components defined by SCAPE, allowing Plato to compose them into a preservation action plan with minimal additional complexity or shim services⁴.

4 Supporting Tools

We provide two additional tools to support the creation and use of components.

4.1 Component Validator Library

We have created a component validator library to support the component repository (see Section 2.1). This library takes the location of a component definition and the location of a profile to check it against, and produces a list of outcomes of evaluating the conditions described in the profile against the component. This includes checking assertions about the cardinalities and depths of ports, and the checking of assertions based on the annotations (both semantic and otherwise) on the component.

⁴ In Taverna terminology, shim services (usually just “shims”) are those that are used to coerce data from the format of the output of one significant processing unit into the format of the input of another significant processing unit. They are typically “uninteresting” from a scientific perspective, but are a necessary adaptation where services are not explicitly designed to work together. One of the things that Taverna components do is conceal any application of shims, allowing the standardization of the data formats in use in a community.

For example, if the profile states that an authorship attribute must be present, the evaluation rule tests whether there is a `net.sf.taverna.t2.annotation.annotationbeans.Author` annotation on the top dataflow in the workflow document described in the component definition. If it is present, the validation rule *passes*, and if the annotation is absent, the rule *fails*.

A third state — *warning* — is used for technical passes for reasons that are unlikely to be desired. For example, if a rule requires checking the depths of ports with a particular semantic annotation, but no such annotations are present.

The validator library produces a list of these assertion-based statements. The API to the validator has a single main Java class, `org.taverna.component.validator.Validator`. This has one principal entry point:

```
List<Assertion> validate(URL componentUrl, URL profileUrl)
```

This takes in the locations of the definitions of the component and the profile (in each case, the URL needs to be to the resource to read that contains the relevant document: for the component, this is to the t2flow, for the profile, this is to the profile definition) and returns a list of evaluated assertions. A long list of exceptions is possible (e.g. if either component or profile is unreadable). The `Assertion` class, which has three trivial subclasses (`Pass`, `Warn`, `Fail`), is just a holder of `final` fields; in particular it contains a description of why the assertion was deemed to have passed or failed, but is otherwise of little interest as it has no externally-callable methods.

The library containing the validator is marked as being executable so that it can be used from languages other than Java by invoking it as a program. When used this way, it takes two command-line arguments (the URLs to the `validate` method described above) and produces a JSON document on its standard output. This JSON document has six keys:

- `allSatisfied` — a Boolean saying whether all assertions succeeded (i.e., are evaluated to either `Pass` or `Warn`).
- `assertions` — the array of assertion statements. Each assertion statement is a JSON object describing the type of assertion (a string from `satisfied`, `warning`, `failed`) and the message string. These are sorted so that the `failed` assertion statements all precede the `warning` assertion statements, and the `satisfied` assertion statements come last.
- `numSatisfied` — a count of how many assertion statements are in the `Pass` state.
- `numWarning` — a count of how many assertion statements are in the `Warn` state.
- `numFailed` — a count of how many assertion statements are in the `Fail` state.
- `numTotal` — a count of the total number of assertion statements.

An example of the output (with extra non-significant whitespace) is:

```
{
  "numTotal":9,
  "assertions":[
    {"message":"no component output port called 'out'","type":"failed"},
    {"message":"ignoring depth constraints","type":"warning"},
    {"message":"ignoring Description requirement","type":"warning"},
    {
      "message":"no Fred activity satisfies semantic constraints",
      "type":"warning"
    }
  ]
}
```



```

    },
    {
      "message": "no depth information for port 'component output'",
      "type": "warning"
    },
    {
      "message": "found DESCRIPTION for component", "type": "satisfied"},
      {
        "message": "found AUTHOR for component", "type": "satisfied"},
      {
        "message":
          "component input port 'jp2file' depth is in permitted range",
        "type": "satisfied"
      },
    },
    {
      "message":
        "confirmed semantic and cardinality constraints for activity(s)",
      "type": "satisfied"
    }
  ],
  "numWarning": 4,
  "numFailed": 1,
  "numSatisfied": 4,
  "allSatisfied": false
}

```

The component validator is currently available as a source checkout from the myGrid Github project, at <https://github.com/myGrid/component-validator>, and uses a Maven-based build process. We plan to use the library in future versions of the Taverna Workbench to support component creation so that the user creating a component can see ahead of time whether they are conforming to the profile of the component family that they plan to publish a component within. However, the validator will continue to also be available as a separate library and executable program as well.

4.2 Component Profile Editor

The component profile editor is a GUI tool for creating and editing component profiles, so that it is not necessary to write these XML documents directly. It provides support for browsing the ontologies in use, so that adding semantic annotation requirements to profiles is significantly easier, and it constrains editing so that only structurally correct modifications can be made.

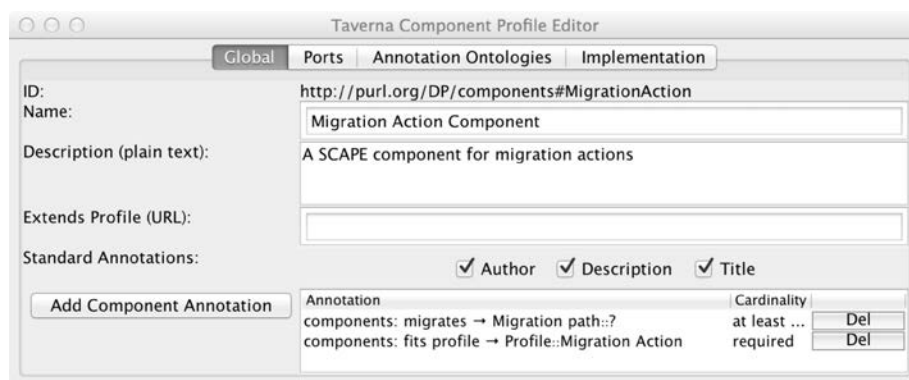
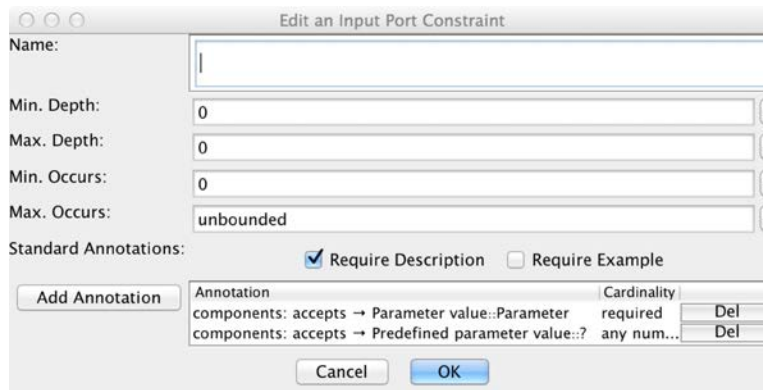


Figure 7: Component Profile Editor showing the SCAPE Migration Action Profile

As can be seen in Figure 7, the GUI provides the capability to control what annotations are present on a component, with those annotation requirements being shown in human-readable form. For

example, the profile requires that a semantic annotation be made using the “components” ontology (i.e., <http://purl.org/DP/components/1.0>, this being configured on the Annotation Ontologies tab) that states that the component matches the semantic model of a migration action as defined in the ontology. It also states that there should be at least one migration path annotated on the component; the combination of these allows Plato (<http://plato.ifs.tuwien.ac.at/plato>) to understand how to use the component when constructing a Preservation Action Plan.



Annotation	Cardinality
components: accepts → Parameter value::Parameter	required
components: accepts → Predefined parameter value::?	any num...

Figure 8: Editing an Input Port Constraint

Similarly, Figure 8 shows the pane used to edit the constraints on an input port. The port in this case is actually one that can occur any number of times (it is completely optional) and which is of depth 0 exactly (i.e., it is always a singleton value) and has some descriptive text on it. The port is identified by the fact that it has an annotation on it that states that this is a parameter to the component, and permits the assertion of possible predefined values that the parameter may be.

The component profile editor is currently available as a source checkout from the myGrid Github project, at <https://github.com/myGrid/component-profile-creator>, and uses a Maven-based build process. We may roll the functionality into a future version of the Taverna Workbench, so that the profile editor tool is an integrated part of the workbench’s component management system.

5 Glossary

Term	Definition
Component repository	A store of <i>Taverna components</i> and <i>component profiles</i> . It is typically expected that the component repository would also be the component catalogue so that the components and their profiles can be found, and it is typically treated as a synonym of a catalogue; myExperiment is both a catalogue and a repository of Taverna components.
Component profile	An XML document that describes the constraints that a <i>Taverna component</i> should adhere to, and the semantic annotations that may be used with that component.
Plan Management Service	A service that holds a <i>Preservation Plan</i> and manages its lifecycle. This can be any service that implements the Plan Management API, described in Deliverable D4.1. Note that a PMS may also implement other APIs and be principally known by other names.

Term	Definition
Plato	A web-based tool that creates a <i>Preservation Plan</i> and provides a user interface for viewing, managing and updating that plan. The plan itself is stored in the <i>Plan Management Service</i> after creation.
Preservation Action Plan	<p>A Preservation Action Plan is part of a <i>Preservation Plan</i> — or a separate document for the purposes of processing — that describes a set of digital objects, an operation (typically a transformation) to apply to each of them, and a rule that allows the determination of whether the operation on a particular digital object was successful on the basis of characteristics measured on the instantiation of the digital object, what it was transformed into, or the comparison of what it was and what it became.</p> <p>A Preservation Action Plan does not describe how to instantiate the digital object, where to archive successful transformations, or where to report the outcome of applying the PAP.</p>
Preservation Plan	A Preservation Plan is a live document that defines a series of preservation actions to be taken by a responsible institution due to an identified risk for a set of digital objects or records (called a collection). It is defined by <i>Plato</i> and stored in a <i>Plan Management Service</i> .
SCAPE Characterisation Components	Characterisation components are a family of <i>SCAPE Components</i> (defined to wrap tools produced in WP9) that compute one or more properties of a <i>single</i> instantiated digital object or file. The output ports that produce measures are always annotated with the metric (in the <i>SCAPE Ontology</i>) that describes what the component computes.
SCAPE Components	SCAPE components are <i>Taverna Components</i> , identified by the SCAPE Preservation Components sub-project, that conform to the general SCAPE requirements for having annotation of their behaviour, inputs and outputs. SCAPE components may be stored in the SCAPE Component Catalogue, which is a part of the myExperiment web service.
SCAPE Migration Components	Migration components are a family of <i>SCAPE Components</i> (defined to wrap tools produced in WP10) that apply a transformation to an instantiated digital object or file to produce a new file. The input is annotated with a term (from the <i>SCAPE Ontology</i>) that says what sort of digital object/file is accepted, and the output is annotated with a term that says what sort of file is produced.
SCAPE Ontology	The SCAPE Ontology is an OWL ontology that formally defines the terms used by computing systems in SCAPE.
SCAPE QA Components	QA components are a family of <i>SCAPE Components</i> (defined to wrap tools produced in WP11) that compute a comparison between <i>two</i> instantiated digital objects or two files. They produce at least one output that has a measure of similarity between the inputs, and that output is annotated with the metric (in the <i>SCAPE Ontology</i>) that describes the nature of the similarity metric.

Term	Definition
SCAPE Utility Components	<p>Utility components are a family of <i>Taverna Components</i> that provide miscellaneous capabilities required for constructing SCAPE workflows, but which are not a core feature of the SCAPE preservation planning process. For example, they can provide assembly and manipulation of XML documents that contain collections of measures of workflows.</p> <p>Note that utility components are not <i>SCAPE components</i> per se; they do not conform to the standard profiles. Instead, they are used in support roles.</p>
Taverna Components	<p>Taverna components are <i>Taverna workflow</i> fragments that are stored independently of the workflows that they are used in, and that are semantically annotated with information about what the behaviour of the workflow fragment is. They are logically related to a programming language shared library, though the mechanisms involved differ.</p> <p>Taverna components are stored in a component repository. This repository can either be a local directory, or a remote service that supports the Taverna Component API such as the <i>SCAPE Component Catalogue</i>. Only components that are stored in a publically accessible service can be used by a <i>Taverna workflow</i> that has been sent to a system that was not originally used to create it.</p>
Taverna Server	Taverna Server is a multi-user service that can execute <i>Taverna workflows</i> . Clients do not need to understand those workflows in order to execute them.
Taverna Workbench	The Taverna Workbench is a desktop application for creating, editing and executing <i>Taverna workflows</i> .
Taverna workflow	A Taverna workflow is a parallel data-processing program that can be executed by <i>Taverna Workbench</i> or <i>Taverna Server</i> . It is stored as an XML file, and has a graphical rendering.
Workflow repository	<p>A service that stores workflows, allowing them to be distributed to other people in accordance with the defined access control policies. A workflow repository that holds Taverna workflows is consequently a Taverna workflow repository.</p> <p>MyExperiment is an example of a workflow repository.</p>