One of the core goals of GAVO-II is to create a VObs data and expertise center at ARI-ZAH. One of its tasks will be to host smaller datasets (images, spectra and catalogues) and/or data set descriptions (metadata) from German institutes who do not have the resources and/or expertise to maintain an online presence themselves. We will develop tools to assist users in uploading their

Fig. 1

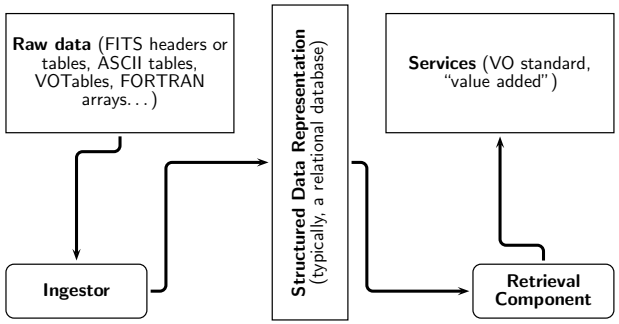

Fig. 2

1. Data Center: Lessons we learned

Markus Demleitner *(msdemlei@ari.uni-heidelberg.de)*

- "Integrated Publishing Infrastructure"
- Metadata
- Ingesting
- Retrieval

The GAVO II proposal in 2005 contained the following:

(vgl. Fig. 1)

This essentially boils down to: "Build a system that takes more or less random data on one end and that spits out both VO-compliant and custom services on the other."

This system is what I call an "Integrated Publication Infrastructure" (IPI). There are a few of these around – at a sufficient level of abstraction, everyone publishing data will have to build or get one. At this conference, some published ones, mostly focused on more or less specific types of data, are presented.

When we set out filling the above lines with life, we found none of the existing packages going in the direction we thought we were headed for, so we started another one. Welcome to the VO.

It turns out that building an IPI is surprisingly involved, and I want to show some of the conclusions we came to, sometimes together with the mistakes that lead us to these conclusions.

2. An IPI from 30 000 ft

(vgl. Fig. 2)

Seen from very far above, an IPI has to fill in the two rounded boxes in the above graph:

- An ingestor that takes the data provided by the scientist, preferably exactly as submitted, and turns it into something that the retrieval component can work with – currently, that's a relational database in most cases, but anything from in-memory data structures to "triple stores" (these are basically what the semantic web people want to organize their data with) could be used.
- A retrieval component that can turn well-defined subsets of the structured data into something the users want. This can be through the DAL ("Data access layer") and VO-enabled tools, the web browser, jabber, or whatever.

In our experience, data providers will not be so wild about having their data "in the VO" as yet that they'll jump through hoops (or even just one) to get their data to you. Also, they'll later come with special wishes, new data, or whatever. Both experiences taken together mean that your ingestor should be able to cope with a wide variety of inputs.

At the other end, unless the data providers have set up web interfaces of their own, the first thing they want is right that, a form based service delivering as much functionality through a web-browser as possible. The retrieval component must at least deliver that, and be reasonably flexible in doing this. VO protocols are a must as well, not so much because currently many data providers ask for it but because it's the Right Thing.

3. Resource Descriptors

Describe a "resource" to the IPI as *declaratively* as possible:

- Where does the data come from, where is it located, who did what to it...?
- How do we get from the input to the structred representation?
- What is the structured representation ("table schema")?
- What formats are available for data output?
- How is the data accessed?
- and more: Privileges, publication, ...

"Declarative" means that you try to define the problem and leave figuring out the solution to the computer. This is in contrast to "imperative", the paradigm of most well-known programming languages. The advantage of the declarative approach is that it usually binds you less to concrete implementations (since the problem typically doesn't change that much), and that becomes important when you have many such descriptions.

On the other hand, for many problems a declarative approach is cumbersome, so for certain problems it's probably not worth the effort trying to work around the truth that in the end you are describing "procedures" (to get from input to output, do this, this, and that).

The trick is to find a way to get by with as few imperative sections ("scripts" in my lingo) as possible.

A word on "resource": RMI says a resource is something "that can be described in terms of who curates and maintains it and which can be given a name and a unique identifier." Since a resource descriptor in our IPI will typically provide more than on unique identifier, this isn't really what we mean by the resource that is described. In the GAVO IPI, what an RD describes is the sum of tables and services arising from one set of source data.

4. RDs: an abridged example

```
<ResourceDescriptor srcdir="lsw">
 <schema>lsw</schema>
 <meta name="referenceURL"
   title="Project Description">
   http://www.lsw.uni-heidelbe...
 <Data sourcePat="data/*.fits"
   id="plates">
  <FitsGrammar qnd="True">
   <Macro name="interpolateStrings"
     destination="instId"
     format="%s, %s"
     sources="OB,TEL"/>
   <Macro name="setSiapMeta">
     <arg name="prodtblKey"
      value="@inputRelativePath"/>
   ...
  <Semantics>
   <Record table="plates">
    <implements name="bboxSiap"/>



   <Field dest="obj" dbtype="text"
    source="OBJ"
    ucd="meta.id;src"/>
 <Service id="siap"
   fieldPath="plates.plates">
  <allow renderers="siap,form"/>
  <publish render="siap"
   sets="ivo_managed"/>
  <meta name="sia.type">Pointed
  <core builtin="siapcutout">
    <arg name="table"
     value="plates"/>
  <srvInput>
   <condDesc predefined="siap"/>
   <condDesc original=".dateObs"/>
  <srvOutput>
   <elgen name="siapOutput"/>
 <Service ...
```

This is a sketch of some aspects of an RD: There is one or more Data sections that define how the structured data looks like (Semantics) and how it is generated from FITS files (FitsGrammar, Macros; these are instructions for the ingestor). There is also one or more Service sections describing how data is delivered to the user.

The details are not that important for now.

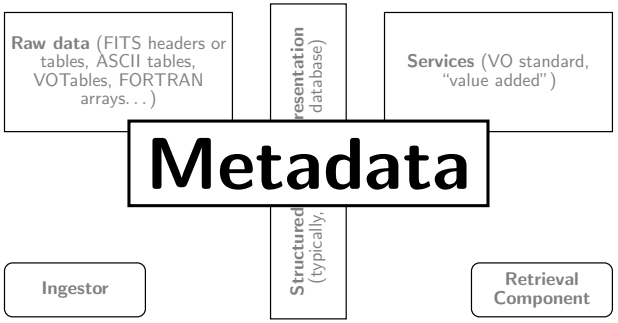

Fig. 3

5. Metadata

(vgl. Fig. 3)

Lesson 1: Metadata is complex, omnipresent, and has many facets. Quite a bit of the VO is about metadata.

Metadata, basically, is data about data, i.e., information that is not directly contained in the data set itself. In the VO, people talk about metadata a lot, not the least because frequently, metadata becomes data itself – think of a database for images that contains pointings, exposure times, observation dates, filters used, etc.

In an IPI, shuffling metadata around is a major challenge, in particular because parts of it have to be shared between various components, while others must not "ooze around". Metadata can be roughly categorized as:

- Column Metadata: Description, UCD, utype, (desired, provided) unit, formatting hints, etc. The VOTable spec helps you here.
- Dataset Metadata: Who made it, who did what to it, where can I get more information, why should I believe it etc. [RMI][1] has more on many aspects of these.
- Access Metadata: Typically machine-interpretable data on how a given service has to be talked to in order to get what result.

As an example for a harmless metadata-related challenge, consider units: You may get a proper motion in rads per century, want to store it as degrees per year (possibly because that makes some calculations easier) but users want to see it as arcsecs per year. So, for a given column you could store three pieces of metadata. We got that one quite horribly wrong.

What you really want is to keep the input unit as metadata of the *ingestor*, not of the column. This is one of the cases where the right way seems obvious except when you are about implementing such a system.

RMI-type metadata comes as a sequence of structured data items:

$$\text{Coverage.Spectral.}\begin{cases}\text{BandpassId}\\ \text{CentralWavelength}\end{cases}$$

$$\text{Coverage.Spectral.}\begin{cases}\text{BandpassId}\\ \text{CentralWavelength}\end{cases}$$

Within the GAVO IPI, we opted for two almost equivalent ways of representing metadata. You can give them in the resource descriptor, serialized as XML, like this:

```
<meta name="creator">
```

[1] `http://www.ivoa.net/Documents/latest/RM.html`

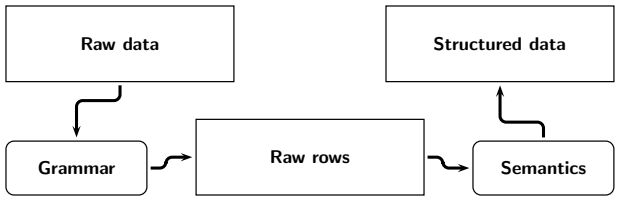

Fig. 4

```
    <meta name="name">Landessternwarte Heidelberg</meta></meta>
  <meta name="subject">astrophotography</meta>
  <meta name="subject">photographic plate</meta>
```

There is also text-based metadata input, e.g. when describing static resources or defaults:

```
creator.name: Landessternwarte Heidelberg
subject: astrophotography
subject: photographic plate
```

In principle, all metadata is string-typed, but certain keys impose additional structure (right now, we only use this to give titles to metadata having URL values. In principle, one should have a rich type system (including, e.g., ranges) on metadata, but it's probably not worth the effort. Yet.

6. The Ingestor

The ingestor's job is to turn any kind of input to strictly structured data, typically in a relational DB.

(vgl. Fig. 4)

Grammars specify how to get fielded data ("strings with labels") from a source (e.g., from FITS headers of tables, from columns of ASCII files, from binary data...).

The semantics specifies how to turn the fielded data into proper, typed, nulled, referenced, etc., data.

The tricky part here is where to draw the lines. The biggest wart in our current design – dating back to the time I thought I'd be writing yet another ingestor – is that the semantics part is largely intertwined with the definition of the structured data, i.e., the field definitions carry information such as the label of the source string or sometimes even the literal form of the strings.

This is bad in particular because (a) it's very hard to manage this kind of information when such fields are copied into, e.g., a definition of the service input or output, and (b) because "macros", preprocessing currently done in the grammar (and rightly so) may already leave non-string content, which confuses matters.

Whatever architecture is better, it's clear that the semantics delivers finished records to the structured data, and the structured data definition has no artifacts of the ingestion process at all.

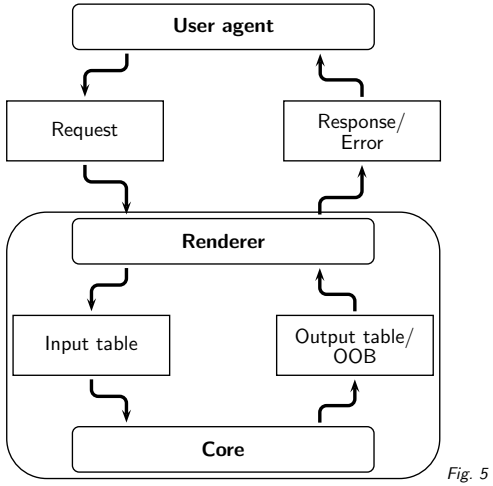

Fig. 5

7. Retrieval

(vgl. Fig. 5)

A service consists of a renderer taking the user input and formatting the response, and a core that does the actual computation.

Renderer and core communicate through standard tables plus potentially out-of-band data (exceptions).

The rationale for this architecture is that the various protocols used to access data may differ widely. A SOAP service, even over HTTP, receives its parameters in a completely different way from a, say, HTTP GET-based service. The result may be formatted as a HTML table, a VOTable or a JPEG.

Things get worse when in comes to error reporting. Even the various DAL protocols the IVOA has defined so far can't agree where in a VOTable an error should be reported. In an "interactive" form-based service you want neither of these but rather error messaging in forms.

Having a renderer handle such details lets one re-use identical cores for all kinds of usage scenarios.

Using "tables" (these are actually data structures designed to carry enough semantics to create VOTables from them) to communicate between core and service is a natural choice for output, where, on success, a table is supposed to be delivered. On input, this choice is less obvious. However, some services require sets of records as input (think an ephemeris service), so this comes in handy as well.

With this basic model, the details are still hard. For example, who decides what columns should be present in the output data? In principle, it should be the core, but really, user preferences can severely influence that decision (e.g., the VERB parameter with DAL protocols).

Actually, for most DB-based queries, the input and output parameters are an important part of the service profile. So, while initially we had placed field selection in the core, we put the "main" decision into the service, and the cores may only override this at their own peril. Since tables carry meta information on the columns contained in them, services and, in particular, renderers can adapt to core's decisions.

Another fine point are input and ouput filters. These process the data when they flow between renderer and core and are extremely hard to get right (e.g., because they again change the field selection and thus have to be part of the core/renderer negotiations). Our current model, in which the service manages these filters on behalf of mainly the core is somewhat suboptimal, but, partly for lack of use cases, we do not have a good solution so far.

8. In Closing

The GAVO IPI is supposed to be Free software. It's not published yet, though, and probably won't be for another year at least, mainly since "end users" will despair on it.

If, on the other hand, you think of starting a VO node and don't mind co-developing: Talk to me.

Live site: http://vo.uni-hd.de