FITS Files and Regular Grammars: A DMaSS Design Case Study

Andrew Cooke*, Alvaro Egaña, Sonya Lowry

August 2006

Abstract

The NOAO DMaSS (Data Management and Science Solutions) Platform has already passed through several development iterations and includes a Database (DB) Loader service. Experience using that service has been mixed.

Recent discussion of the proposed Metadata service focused on 'vocabularies'. We realised that we could treat FITS headers as a formal language and that this language was described by a regular grammar. It then became clear that part of the DB Loader's configuration was, in fact, a regular expression (expressed in several files of rather verbose Spring XML!) Furthermore, a separate (and equally confusing) configuration was now clearly related. And we could extend things further: the same data structures could help detect 'almost duplicate' data reliably.

Reviewing our results in terms of the system architecture, we suggest moving newly identified responsibilities to appropriate services. The end result is a cleaner design, better integrated within our architecture, and in which we have much more confidence. What started as an idle theoretical curiousity yielded very practical results.

Contents

1 The Problem **1**
- 1.1 FITS Headers . . . 1
- 1.2 NOAO DMaSS . . . 2
- 1.3 Vocabularies . . . 2
- 1.4 Types and Values . . . 2

2 The Analysis **2**
- 2.1 Formal Languages . . . 2
- 2.2 Simple Model . . . 3
- 2.3 Reintroducing Values . . . 3
- 2.4 Graph Structure . . . 3
- 2.5 Semantics, Ontology . . . 4

3 The Solution **4**
- 3.1 Recognition Trees . . . 4
- 3.2 Reality Check: Workflow . . . 4
- 3.3 Categories, Orthogonal Vocabularies . . . 4
- 3.4 Separation of Concerns . . . 5
- 3.5 Implementation . . . 5
- 3.6 Reality Check: Field Map . . . 7

4 Conclusions **7**
- 4.1 Vocabularies . . . 7
- 4.2 Loading and Remediation . . . 7
- 4.3 Duplicate Data . . . 7
- 4.4 Theory . . . 7
- 4.5 Problems . . . 8

5 Addendum I **8**
- 5.1 Non-Unique Fields . . . 8
- 5.2 Hierarchical Headers . . . 8

6 Addendum II **8**

*andrew@acooke.org

1 The Problem

1.1 FITS Headers

Scientific data often consist of two components: a collection of 'measurements' ('science data'; e.g. an image) and a 'description' of those values ('science metadata').

In the most common file format, FITS (Flexible Image Transport System[1]), the science metadata are stored as name/value pairs (grouped in 'headers'). Apart from multi–extension (MEF) files (which we do not believe significantly complicate the problem, and which we will ignore here), the metadata have very little structure: the names form a simple set, with no repetition or nesting, and almost no ordering.

1.2 NOAO DMaSS

The NOAO DMaSS[2] is a set of services that can be deployed as an archive for astronomical data. Early iterations already include a DB Loader service, responsible for loading science metadata into an SQL database.

Experience with the DB Loader has identified problems with the configuration, which is distributed across several different sets of files: a workflow[3], which identifies particular data products (for example: images observed on a certain camera/telescope); a mapping of metadata names to database fields; and a description of the database. Only the last of these is generated automatically, the others must be carefully constructed by hand[1].

1.3 Vocabularies

Recent discussion within NOAO has focused on the concept of 'vocabularies' — the *types* of science metadata expected for different data products. NOAO manages many instruments/telescopes, and the DMaSS is intended as a general system, so we must be able to manage a wide variety of vocabularies.

Since science data enter the system as FITS files the range of vocabularies reflects the range of headers we must handle. Further complexity is added by the evolution of vocabularies over time and modifications/additions to the metadata by various data–handling systems.

[1]This service currently loads database fields (SQL table columns) directly. Changing the system to bind values in a set of objects, managed by the Metadata service, is anticipated in the design, but would not simplify the problem addressed here.

1.4 Types and Values

Faced with a wide range of vocabularies, we needed a way to structure our problem. One source of complexity was the interplay between types and values: the set of names that a FITS header might contain depended on the values associated with some of the names.

While this is completely obvious to anyone who has worked with FITS data, it is easy to see how it can become difficult to manage in a software system. The same dependency within a naive OO design would give objects whose class changes depending on the values contained. While this is possible with predicate classes[4] it cannot be expressed directly in a language with a fairly simple, static type system like Java or C++[2].

On the other hand, this didn't seem like a 'hard' problem. After all, lots of people write programs that manage FITS data! We were tempted to ignore the issue and simply 'do the work', but then stumbled across an argument that led, in a sequence of simple steps, to a solution that combines decent engineering with a consistent logical framework.

2 The Analysis

2.1 Formal Languages

In the following sections we use a few ideas from Formal Languages[5], but we do not rigorously prove any results; our aim is to use the theory to help illuminate a practical design. While we do not expect 'real life' to behave in such an idealised manner, we do hope that the argument will help us understand how best to deal with exceptions and irregularities.

This theory is not particularly obscure; it is probably familiar as the basis for parsing, regular expressions, etc. We will use the following ideas: that a *language*[3] is composed of an *alphabet* (a list of *words*) and a *grammar*; that *grammars* define how *words* are

[2]This mixing of types and values also appears to be related to Russell's paradox in set theory.

[3]We use *italicised* text to indicate that the word is being used in a formal manner, as the meanings do not coincide exactly with common use.

combined to produce valid *sentences*; that *grammars* can be categorised in various ways; and that perhaps the simplest and most common of these is a *regular grammar*, whose *sentences* can be described by regular expressions.

2.2 Simple Model

The metadata in a FITS header are, for the most part, unordered. We start by imposing an arbitrary order (e.g. sorting their names alphabetically).

To further simplify our initial model we will associate each metadata name with a different letter and ignore the associated values. This gives us our *alphabet*. Valid headers, sorted and represented in this manner, form *sentences*.

With these assumptions, example FITS headers might be `abcde`, `abpqrs`, or `xyz`.

We can match those examples using the regular expression `(ab(cde|pqrs)|xyz)`. Since we could write a similar expression for any header (*sentence*) in this model we have shown that they are described by a *regular grammar*.

This result is not very surprising. The next level of complexity is a *context free grammar*. These are characterised by nested structures, but FITS headers are not that complex (they lack any idea of 'namespaces' for example).

2.3 Reintroducing Values

We assumed that the we could represent metadata as letters. This is reasonable if we do not care about the value. However, there are at least two cases which require more care.

First, as mentioned in section 1.4, there are some metadata whose values affect the choice of vocabulary. For example, the instrument type. In such cases we must associate a different *word* in the *alphabet* with each distinct option. If originally we associated the name `INSTR` with the *word* `a` we must now introduce distinct *words* for each type of instrument. `INSTR='MOSAIC'` might be mapped to $\mathtt{a}_1$, `INSTR='ODI'` to $\mathtt{a}_2$, etc.

Second, we might want to validate the values. It is trivial to extend the argument to include regular expressions for each value, but perhaps validation requires a more complex function. Technically a *grammar* is *generative* — it can be used to construct valid sentences — but in our case we are using the *grammar* only to recognise (not construct) valid sentences. In such cases, we can use an arbitrary predicate as a *word*, providing we do not use any information from 'outside' the value in question.

The second case may also apply to the first. There might be some metadata that affected the choice of vocabulary via a complex function. Provided this function takes only the single metadata value as an argument we can extend our *alphabet* with a suitable predicate[4].

2.4 Graph Structure

The regular expression `(ab(cde|pqrs)|xyz)` from section 2.2 was constructed by grouping common prefixes. Matching a FITS header against the expression involves identifying a path through a tree, starting at the root (leftmost node) and ending at one of the leaves (`e`, `s` or `z`). We can consider the regular expression as a 'decision tree' whose leaf nodes classify FITS headers.

This tree–like structure is intuitive and easy to represent in software. And the leaf nodes suggest a process for identifying vocabularies. In general, however, regular expressions can be more complex — they are equivalent to finite state machines. So we would like to understand whether this tree structure is (1) sufficient and (2) practical.

A state machine can be represented as a directed graph (DG) whose edges are transitions from one state to another. Each edge is associated with matching a *word* in the *language*. A DG may contain cycles, but we have already noted that *words* in our *sentences* do not repeat (every name in a FITS header is unique; there is a single namespace). So cycles (repetition) do not seem to be necessary.

Without cycles, we have a directed acyclic graph (DAG). This is still more general than a tree; the branches of a tree cannot 're–join'. How significant is this lack of expressivity for trees? For finite

[4] In extreme cases we might even group several metadata together and treat them as a single value (e.g. the WCS).

length sentences we can enumerate all possible paths through a graph, giving a tree. So for any DG we can always construct a tree–like regular expression that identifies the *language* (at the expense of duplication of information). Since DAGs are a subclass of DGs the argument also applies there. A tree is sufficient, but not optimal.

We will return to this later when we discuss 'orthogonal vocabularies'.

2.5 Semantics, Ontology

Our initial analysis assumed (section 2.2) an alphabetical ordering of names. Optional metadata names near the start of the alphabet will introduce structure in our tree–like regular expression (they will add unnecessary branching near the root). Our *grammar* is very sensitive to 'noise' in the FITS header.

We have not relied on the type of ordering in any of our arguments above and are therefore free to choose one to address this problem. We will impose an order based on our intuition about 'importance'. The introduction of a semantic ordering will give a more compact representation[5].

This is (again) an argument that sounds opaque within our semi–formal approach, but which is 'obvious' in practice. We are suggesting that the regular expression should start with metadata common to all valid FITS files[6]. Next should be 'important' fields (`INSTR`, for example, and then perhaps `OBSTYPE`) that identify different 'types' of FITS files.

In this way, our decision tree reflects the meaning within the metadata; the leaf nodes classify FITS headers within some ontology.

3 The Solution

3.1 Recognition Trees

We call the semantic decision tree described above a 'recognition tree'. Recognition trees are a tool to organise our knowledge of the structure of FITS headers and classify science metadata.

The application of a recognition trees to a FITS header mirrors the use of a regular expression to match a *sentence*. Starting at the tree root we repeatedly select a child node based on matching the science metadata. Eventually we either fail to match — the FITS header is malformed or we have an incomplete tree — or we arrive at a leaf node.

Each edge in the tree corresponds to a predicate that tests science metadata. Each predicate specifies a single name and an optional test on the associated value. The test need not be a regular expression, but must base its decision only on the value given.

The structure of a recognition tree is influenced by our knowledge of the semantics associated with each predicate. This allows us to associate leaf nodes with ontological concepts (e.g. to classify the FITS file as a certain data product).

3.2 Reality Check: Workflow

The workflow in the current DB Loader is, in retrospect, a simple recognition tree. The first edges are tests on the `INSTR` metadata value and the leaf nodes identify different data products (Mosaic sky flats, generic bias frames, etc).

This workflow is a hand–written, somewhat ad–hoc, Spring XML configuration. Our analysis shows that we can safely (ie without risk of using too restrictive a representation) move this definition to a tree–like structure in the database. The emphasis will change from 'DB Loader configuration' to a more specific 'Vocabulary Service'.

3.3 Categories, Orthogonal Vocabularies

The introduction of a separate service, any associated improvement in the user interface, and a better understanding of the role of this information, will all lead to a more detailed recognition tree with a finer granularity in the ontology. Rather than a simple 'Mosaic sky flat', we will have categories like 'Mosaic sky flat using WCS projection XXX with additional

[5] This is, in some sense, the definition of a 'useful' ontology.

[6] Later (section 3.4) we will restrict the use of this tree to identifying vocabularies; metadata common to all can then be discarded.

information from v1.2 of the Kitt Peak Data Handling System'.

For many applications this is 'too much information'. We must provide broader classes that group related leaf nodes. A practical system will probably support many different, overlapping categories. These different categories will probably correspond to different parts of our data model.

This fine level of detail in the leaf nodes is also driven partly by the use of a tree. We noted in section 2.4 that a DAG allowed branches to re–join. The structure we have chosen leads to a proliferation of branches.

An alternative solution to this problem might be to separate orthogonal issues into separate trees. For example, consider WCS–related header properties. We may want to detect different WCS representations, even though these variations have little bearing on the classification into different data products. This could be separated into a second, WCS–specific decision tree.

This separation appears to be possible even when there is some overlap of concerns. For example, we might include WCS coordinate type in both the WCS vocabulary and the science data product vocabulary (where it would be checked for consistency with imaging or spectral data).

Or perhaps all three approaches (categories, orthogonal trees and DAGs) are complementary. This is still an open problem — we will probably implement a single tree with categories initially and then revisit the issue in a later development iteration.

3.4 Separation of Concerns

There is a clear parallel between the Vocabulary service and a parser. Both use grammars to recognise and validate data. A parser also returns values. This raises the questions 'how much validation should the Vocabulary service do?' and 'to what extent should the Vocabulary service be responsible for extracting values from data?'

To answer these questions we need to consider the Vocabulary service within the context of the system architecture. Our architectural definition is:

A Vocabulary is a translation between the system metadata (as surfaced by the appropriate service) and an external syntax. The translation may be in either or both directions.

It is easy to imagine examples (e.g. when the external syntax is conversational English, and the vocabulary service is used to provide 'helpful names' for system metadata) where the Vocabulary service is little more than a table that associates the appropriate syntax with system metadata fields.

So a possible separation of concerns would be to use the recognition tree to select a vocabulary[7] for a given FITS file and then provide translation through appropriate cross–references.

These 'translations' are a natural point at which to store additional functionality that is specific to a metadata value: validation instructions; Java type used for encapsulation, etc.

The Vocabulary service is then used in two stages. First, the relevant vocabulary is recognised. Second, the appropriate vocabulary is provided, with appropriate translation information provided for each element in the data model[8]

This reduces the amount of detail needed in the recognition tree — only properties at branch nodes are required — and will reduce the complexity discussed in the previous section.

3.5 Implementation

Figure 1 sketches a possible implementation of the ideas described above. The recognition tree is on the left; the vocabulary in the middle; and the data model (part of the system metadata) on the right.

A Vocabulary is composed of 'translation sets' that correspond to classes in the data model; a bean path[9] identifies the appropriate class and, together with the

[7] We group vocabularies that are related by a single recogniser within a 'vocabulary family'.

[8] One limitation of this analysis is that it only handles data with a name/value structure; however, this is itself a broad class.

[9] This discussion assumes that the data model is instantiated as JavaBeans[6]. A bean path is a dotted sequence of names that select a path through linked objects (typically translated to a sequence of calls to 'getter' methods).

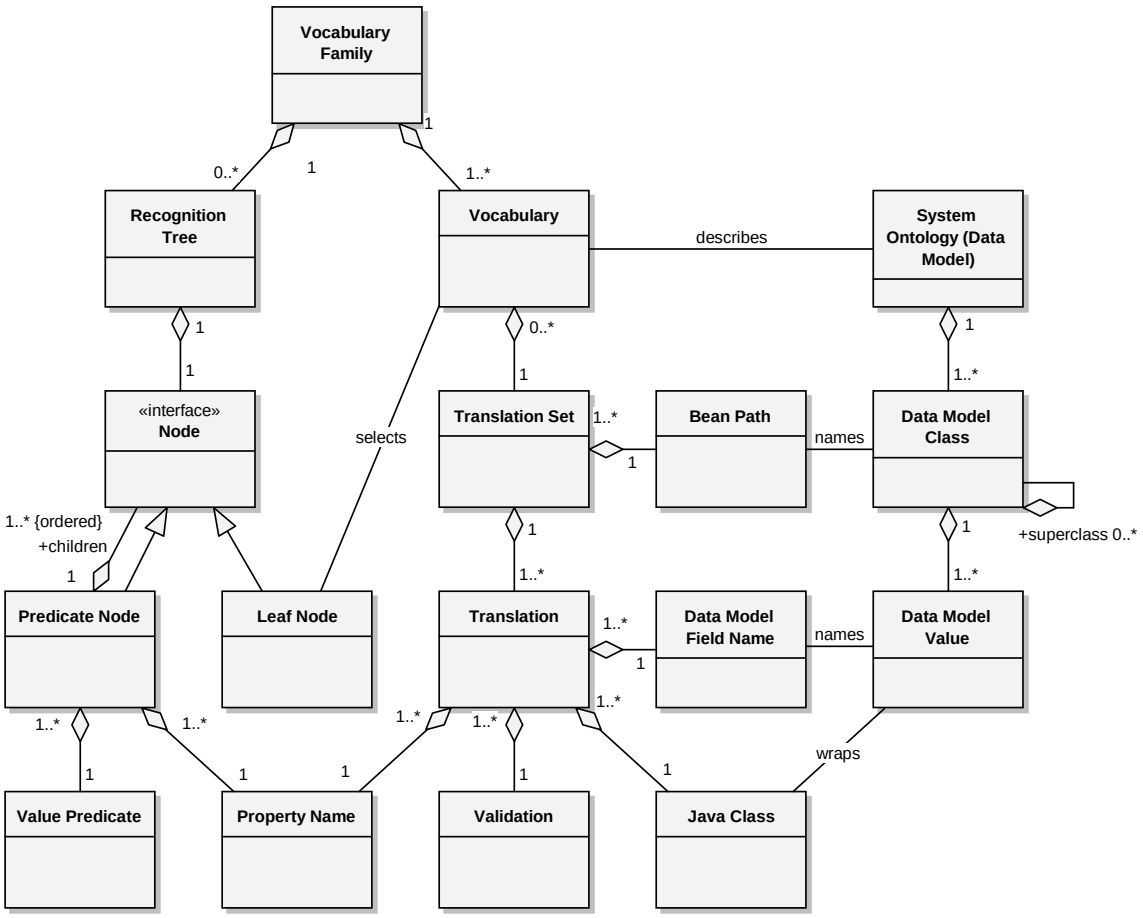

Figure 1: A possible implementation of the ideas discussed here. Details of the data model are intentionally vague.

```
HIERARCH ESO DET WIN1 STRX =   3 / Lower left pixel in X
HIERARCH ESO INS FILT1 NAME = 'OIII/3000' / Filter name
HIERARCH ESO OBS NAME =  'NGC 1275' / Observation block name
```

Figure 2: Example ESO "hierarchical" headers.

field name for a particular translation, would allow dynamic access to values.

This solution will probably evolve further. For example, it is probably incorrect to assume that there is always a 1–to–1 connection between the metadata items in the external syntax and those in the internal model.

3.6 Reality Check: Field Map

The 'translations' correspond to the mapping from science metadata to database (or object) fields that is configured in the DB Loader (section 1.2).

Our experience with the DB Loader suggests that many translation sets will be shared between several vocabularies.

4 Conclusions

4.1 Vocabularies

We started with very little understanding of how to apply the concept of vocabularies to FITS header properties. We now see that each distinct combination of 'interesting' metadata names identifies a separate vocabulary. Since FITS files vary widely we have many vocabularies and need an efficient approach to managing them. This work separated into two related steps.

First, we will use a recognition tree to identify the vocabulary in question. In practice we may split vocabularies into several orthogonal (sub-)vocabularies; we can use separate recognition trees for each, or we can use a single tree whose leaf nodes are categorised by vocabulary.

Second, we structure vocabularies to match our data model. For each field in the data model, we store appropriate information in a 'translation'. These are grouped in 'translation sets' that mirror the classes in the data model.

4.2 Loading and Remediation

In retrospect, the code we developed in an earlier iteration to remediate data and load the database, already contained the structures we describe above. However, we had a poor understanding of the logic that fixes the underlying relationships.

For example, we constructed (the equivalent information to) vocabularies with a text–based configuration that 'imported' related files. The import mechanism was general and poorly constrained. We now understand that by restricting the configuration to reflect the structure in the data model (and moving this information into the database) we can make the relationship between vocabulary and data model explicit, avoid inconsistencies, and simplify the management of these relationships.

4.3 Duplicate Data

An interesting additional use for the ideas identified above is in the detection of 'practically duplicate' data: we would like to be able to identify duplicate copies of the 'same' data, even when some header values change. To do this reliably we must distinguish between metadata which are meaningful, and those which are 'noise' (e.g. time of entry into a data transport system).

One solution to this would problem would be to mark certain translations as 'meaningful' and use only the science metadata that they select. Furthermore, one way of defining 'meaningful' is to use metadata names that appear in the recognition tree.

4.4 Theory

We are interested in how best to use ideas from 'theoretical' computer science in practical programming[10]. While we see many interesting ideas we are also acutely aware that we must avoid self–indulgent hand–waving that gives no practical results. Since we feel that this analysis was productive we are somewhat encouraged to try a similar approach in the future.

One reason for the success here, we believe, was an iterative approach to development. Earlier code helped identify problems and motivate our work. It

[10]As is probably obvious, we are somewhat sensitive about our approach, feeling trapped between theoreticians, who will see very little theory here, and those programmers who believe every problem can be solved by using someone else's toolkit.

was also reassuring to find concrete support for our new ideas — albeit in a rather rough form — already implemented in our software.

4.5 Problems

A further advantage of this analysis is that we can identify possible problems by checking our assumptions against reality. One obvious difficulty is the handling of FITS properties which are 'sequential fragments'. This occurs with WCS[1] encodings, for example, which exceed the maximum length for a single header field[11]. The solution in this case is probably a pre–processing step that generates a single, concatenated value when parsing incoming metadata (and a corresponding fragmentation step when generating a header).

5 Addendum I

This section was added after my ADASS talk in October 2006. It addresses questions raised by the audience.

5.1 Non-Unique Fields

Rob Seaman pointed out that some files contain duplicate fields; it seems that these are badly formed files from non-compliant software/bugs. This raises an issue I did not address — verification.

The information stored in the vocabulary service can help do three things: identify headers; verify headers; extract information from headers. Only the first of these is covered in any detail here. Rob's example shows that verification should occur before identification (and implies that some remediation may also be necessary, if these files are to be ingested).

5.2 Hierarchical Headers

Dick Shaw raised the (non–standard) use by ESO of "Hierarchical Keywords". The ESO documentation[7] gives the example shown in figure 2.

While it is true that a regular grammar cannot handle indefinite hierarchies, the scheme chosen by ESO is sufficiently restricted that it would work within the framework described here. It is only necessary to treat the whole name (including spaces) as a single string.

[11] Based on punched cards

6 Addendum II

This section was added after I found an old copy of 'Parsing Expression Grammars: A Recognition–Based Syntactic Foundation[8]'. That paper gives a much clearer explanation of the difference between a generative grammar and a 'recogniser' (section 2.3), shows that hierarchical expressions can be handled if only recognition is required (as here), and gives references to related, useful work.

References

[1] FITS Documentation
http://fits.gsfc.nasa.gov/fits_documentation.html

[2] S Lowry 2006 (ADASS, in preparation)

[3] A. Cooke 2006; A Tiny Workflow in Spring
http://www.acooke.org/andrew/papers

[4] C, Chambers 1993; Predicate Classes, Lecture Notes in Computer Science
http://citeseer.ist.psu.edu/chambers93predicate.html

[5] D. Grune, C. Jacobs 1990; Parsing Techniques — A Practical Guide
http://www.cs.vu.nl/ dick/PTAPG.html

[6] JavaBeans
http://java.sun.com/products/javabeans/

[7] The ESO Data Interface Definition
http://archive.eso.org/dicb/

[8] Parsing Expression Grammars: A Recognition–Based Synctactic Foundation
http://pdos.csail.mit.edu/papers/parsing:popl04.pdf