Javadoc like technical documentation for CAPRI

- by Wolfgang Britz, July 2008 -

Introduction and background

Since 1996, CAPRI has grown to a rather complex (bio-)economic modelling system. Its code based includes hundredth of single GAMS files, and ten thousands of lines. Not only newcomers face the challenge to get an overview about dependencies in the huge code base and to link the technical implementation to methodological concepts and documentation. On top, the large-scale character of CAPRI often asked for technical features in the GAMS code which are far from the solution chosen for tiny examples as often presented in courses, as the wide spread usage of dynamic sets, conditional includes, the usage of $batcinludes or the application of the grid solve feature.

The task of documenting and keeping an overview of the CAPRI code base is certainly not eased by the fact that basically any object in GAMS has global scope. The concept of functions of subroutines underlying many other programming languages with clearly defined lists of variables passed in and out is not implemented in GAMS. Encapsulation and modularisation are hence not naturally supported by GAMS. That also renders automated documentation of the code more challenging compared to other languages.

Since quite a while, CAPRI user community discusses about some refactoring of the code base on more clearer coding standards with the aim to ease code maintenance, documentation and further development. That refactoring should also cover standard for in-line documentation, including a better link to the methodological documentation. The project CAPRI-RD which will most probably start in spring 2009 will attack some of these tasks in specific working packages. But clearly, that will only become success if the underlying concept is generally accepted and implemented by the community of CAPRI developers. That means that the value added of following coding and documentation standards must be visible to any developer.

The paper is thought to open the discussion and trigger feedback about how to generate an easy to maintain and useful technical documentation for CAPRI, based on the example of JAVADOC (http://de.wikipedia.org/wiki/Javadoc). It is organized as follows. The next short paragraphs will list desired properties of a technical documentation for CAPRI, followed by a

more detailed discussion of a proposal for an implementation which is working as a prototype. The last chapter will then show selected screenshots.

Desired properties

The main properties a technical documentation of CAPRI should fulfil are as follows:

- Avoiding redundancies, i.e. information should whenever possible only inputted once. Specifically, selected comments added in-line in the code should be ported over to the technical documentation.
- Changes in the code structure should possibly be reflected automatically
- The documentation must be able to reflect different GAMS projects (CAPMOD, CAPREG ..) and to differentiate between instances of the same project (CAPMOD in basline or calibration mode …).
- Its biggest part of the technical documentation should be constructed directly from the code based in an automated way.
- It should also collect information from the SVN versioning system

Proposed implementation

The main ingredients of the proposed implementation are as follows:

- The final format of the technical documentation is based on automatically generated static HTML pages, following the example of JAVADOC, with some JavaScript to allow for collapsible trees
- The methodological documentation will continue to be edited in Word, and converted into a PFD-document. It will comprise references to GAMS sources (individual GAMS files) or even GAMS objects (variables, equations, models, parameters). Those references can be addressed in the GAMS code, and the HTML pages will allow opening the PDF-document at the referenced point.
- As with JAVADOC, technical documentation should be edited as in-line comments into the GAMS sources, based on clear in-line documentation standards. Each GAMS source as a file header with standard properties about the file.
- In-line documentation will be mostly based on two levels: the level of individual GAMS files and on the level of individual GAMS objects. In some cases, that may

require to break down larger programs in smaller pieces, with a clear task and eventually clear inputs and outputs.

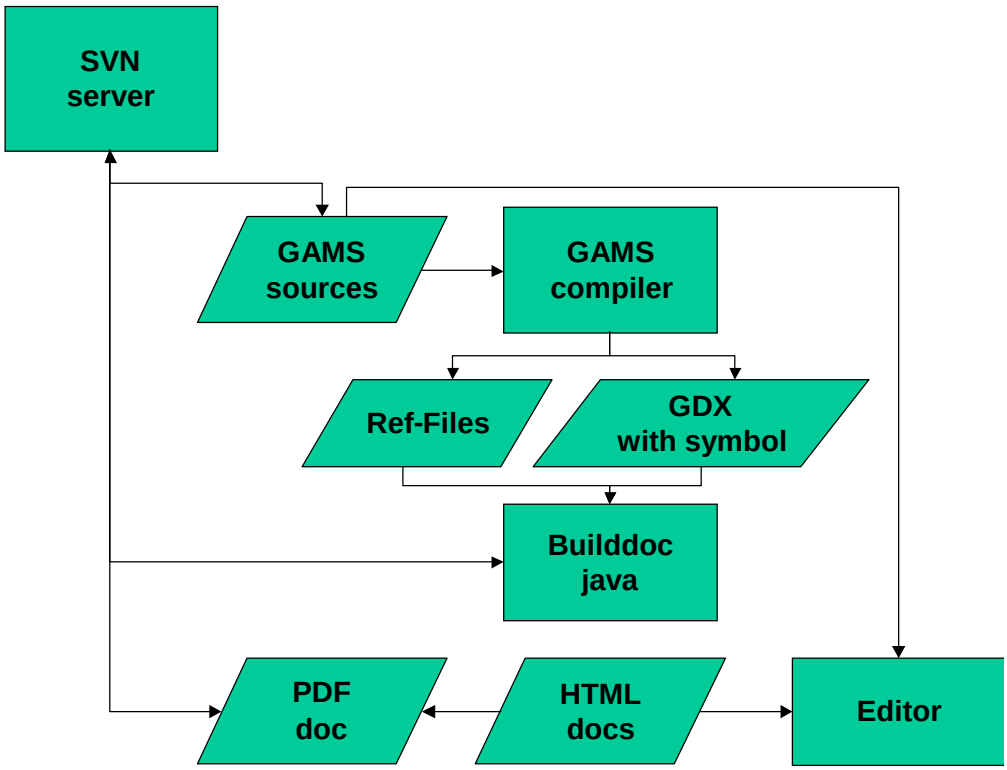
Technical implementation

Overview

The following diagram depicts the general approach. The SVN server will host the GAMS sources, the documentation builder (Builddoc) as a Java application and the PDF with the methodological documentation. Users synchronize their local work copies with the server. In order to avoid developing in Java a new parser for GAMS code, the GAMS compiler itself is used to generate the necessary input for the technical documentation. Two different types of files for each “project” or “instance” included in the documentation are used for that purpose so far:

1. So called **“REF” files**, which list information in which files and in which line symbols are declared, defined, assigned and referenced. They also comprise information about long texts and domain of the symbols. The “REF” file can be generated by the argument rf=*filename* when GAMS is called (e.g. ”GAMS capmod a=c rf=capmod.ref”). As the GAMS compiler itself is used, conditional includes and the like are automatically treated as during execution time. That opens also the possibility to include the generation of the documentation in the GUI.
2. **GDX files generated with an empty symbol list at compile time** ($GDOUT module.gdx; $UNLOAD; $GDOUT). The resulting GDX file will comprise all sets, parameters etc. used by the programs, and most importantly, the set elements as declared. The name of the GDX file could be passed as a parameter by the GUI.

Those files hence reflect the actual local code base with any local modification, and can be generated for a specific instances of the projects (e.g. for CAPMOD with and without the market module etc.). A JAVA application named *Builddoc* parses both types of files, on demand for several projects, and generates static HTML pages. The GAMS code comprises in-line comments carrying information about references to the methodological documentation, and the HTML pages comprise calls to the editor to open the actual source code at the local machine, as well as information about relation between the different GAMS Symbols.



Handling of GDX files

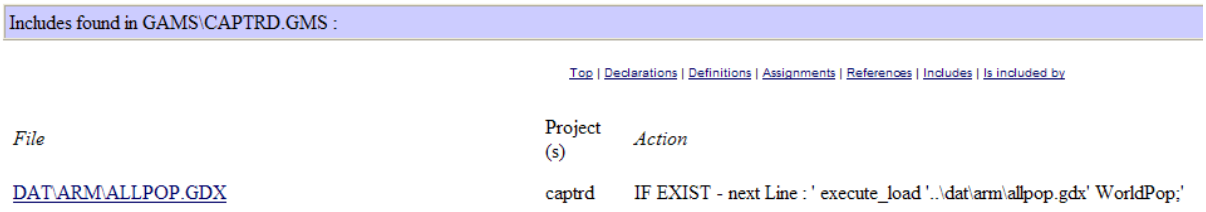
The “expand” option generates information about GDXIN and GDXOUT statements as those are executed at compile time. Consequently, files addressed via GDXIN or GDXOUT are automatically reported in the documentation system.

Hovever, the file does not comprise information about the “execute_load” and “execute_unload” executed at run time. That is quite clear, as the statements may be comprised in program structures as loops or if statements where there are never reached at execution time. We need hence a work around to report those files in the documentation system if we would avoid writing a new GAMS parser.

However, “$IF EXIST” statements are taken into account by the expand command. It is therefore proposed to put an “$IF NOT EXIST” combined with an abort statement before all “execute_load” statements. As seen in:

```
$IF NOT EXIST ..\dat\arm\allpop.gdx $ABORT ..\dat\arm\allpop.gdx is missing
 execute_load '..\dat\arm\allpop.gdx' WorldPop;
```

By doing so, the program will already at compile exit if one of the necessary files is missing. That avoids starting a process and eventually overwriting files which then will stop later due to missing input data. The HTML page will report that sequence as:



Includes found in GAMS\CAPTRD.GMS :

Top | Declarations | Definitions | Assignments | References | Includes | Is included by

File	Project (s)	*Action*
DAT\ARM\ALLPOP.GDX	captrd	IF EXIST - next Line : ' execute_load '..\dat\arm\allpop.gdx' WorldPop;'

The use of "$IF EXIST" in the context of "execute_unload" can only be motivated with the fact to produce code which is better documented. Here, is it proposed to warn the user at run time about the fact that the file is overwritten.

Structure of the HTML pages

There are basically two types of HTML pages:

1. *Pages for individual objects* (parameters, sets, variables, equations, models, acronyms, functions, files and source files)
2. *Summary pages for classes of objects*, per project in alphabetical order. An additional page lists all set elements.

The *pages for the individual objects* carry the following information:

- Name of the object (e.g. DATA) and type (parameter, set, variable etc.)
- Long text description as given in GAMS declaration
- Domain information, as hyperlinks to the domain sets.
- In which files and for which projects (as capmod, capreg …) the object is declared, defined, assigned and referenced.
- In the case of sets: derived subsets, and objects where the set is used as a domain. Elements of the sets and the subsets.
- In the case of source files: which symbols are declared, defined, assigned and references in the files. Information from SVN (version, local modification,s out-of-date with server). Included files, and files which include the file. For GDX files: where included and included by which file.
- "Tagged" in-line comments taken from the source code files, what is called "doclet" (see e.g. Sun document about how to write Doc comment for JavaDoc) in JAVADOC, see .e.g. wikipedia article

Tagged in-line comments

Similar to the element comments underlying JAVADOC (see e.g.), “tagged” in-line comments are proposed for the inline code of CAPRI (sometimes called “doclets”, e.g.) . The following shows a possible implementation which is currently already operational:

…

* @start

* @author W.Britz

* @docRef perfect aggregation of production

* @seeAlso gams\capreg\cons_levels.gms

MODEL CONS_LEVLS / …

..

In the example above, the REF file will comprise the information were the model CONS_LEVL will be declared, and the JAVA application will search backwards for lines with tags (@..). Those tags will be linked to the object, and integrated in the HTML pages. The @start tag must be used to declare the start of the documentation for the current symbol.

Refactoring Consequences for CAPRI Gams Code

1. All files should carry a header which reports the purpose of the file, and if possible, an author (contact person). The file header should start with a line of stars and end with a line of stares. All lines in the file header should start with a “*”.
2. The use of *$GDXIN* is discouraged as it may load in huge amounts of data at run-time. Equally, it will load element codes comprised in the data sets even if they are not referenced later in the code. The only exemption is when the symbol must be loaded at run time as in case of META data, instead, execute_load should be used.
3. An “$IF NOT EXIST *myFile* $ ABORT *myFile* is missing” statement should be in the line before “execute_load *myFile someSymbols*”.
4. An “$IF EXIST *myFile* $ LOG *myFile* will be overwritten” statement should be in the line before execute_unload *myFile someSymbols*”.
5. All symbols should be declared with a clear long text description, i.e. statement in the style “SET A;” are discouraged.

6. Code in lengthy files as CAPREG or CAPMOD should be moved into new files which are included so that a more modular structure is evolving. The new file should have a clearly defined and encapsulated task which is described in the file header.
7. Symbol declaration should where necessary be preceded by a “doclet” of the form
   * @start or, alternative, a blank line
   * @DocRef reference to the methodological documentation (optional)
   *@ seeAlso reference to other file or symbol (optional)
   * Any comments
   Declaration (as SET A “The alternative technologies per production activty” / T1,T2 /;
8. Symbols, especially when they are not widely used across programs should carry meaningful names.

Other recommendations arising from analysing the files are:

1. Single lines in the code should not exceed the size of a normal screen width when using medium sized fonts.
2. Indentation should be used to render the program structure defined by loops, if statements and the like more visible.
3. Especially tricky statements which use complex $ operators, several cross-sets and the like should be preceded by some explanatory comments.
4. Symbols which are only used locally in a file should be deleted from memory by “option kill= …”.[1].
5. Before defining a new set one should check if not the very same collection of elements is not already defined.
6. Lengthy data tables should be moved into a gdx file to reduce the number of code lines.
7. Data should be accompanied by meta data.

Clearly, the standards and recommendations require further discussion inside the network, and must become part of a programming guide.

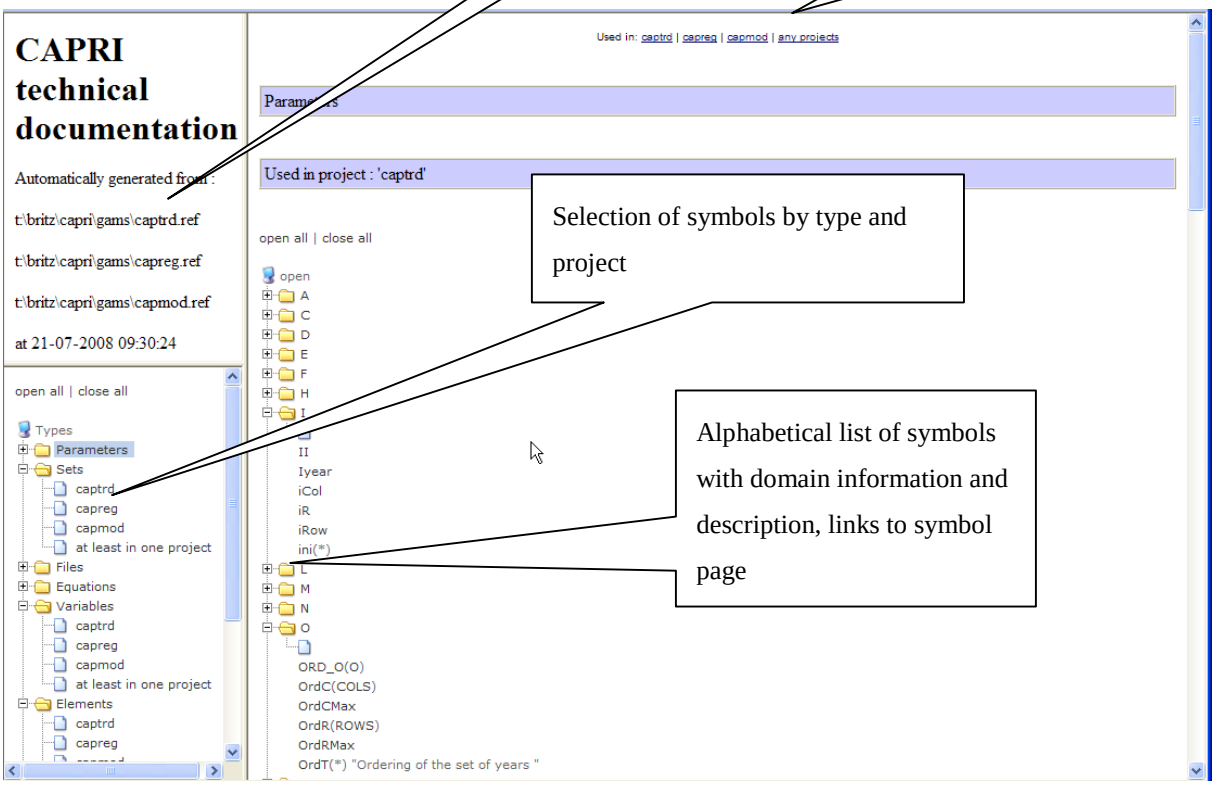
---

[1] A feature request was sent to GAMS to support local scope, so that a symbol can be declared local for a file and subdirectory, and the compiler will raise an error when it is used out of scope.

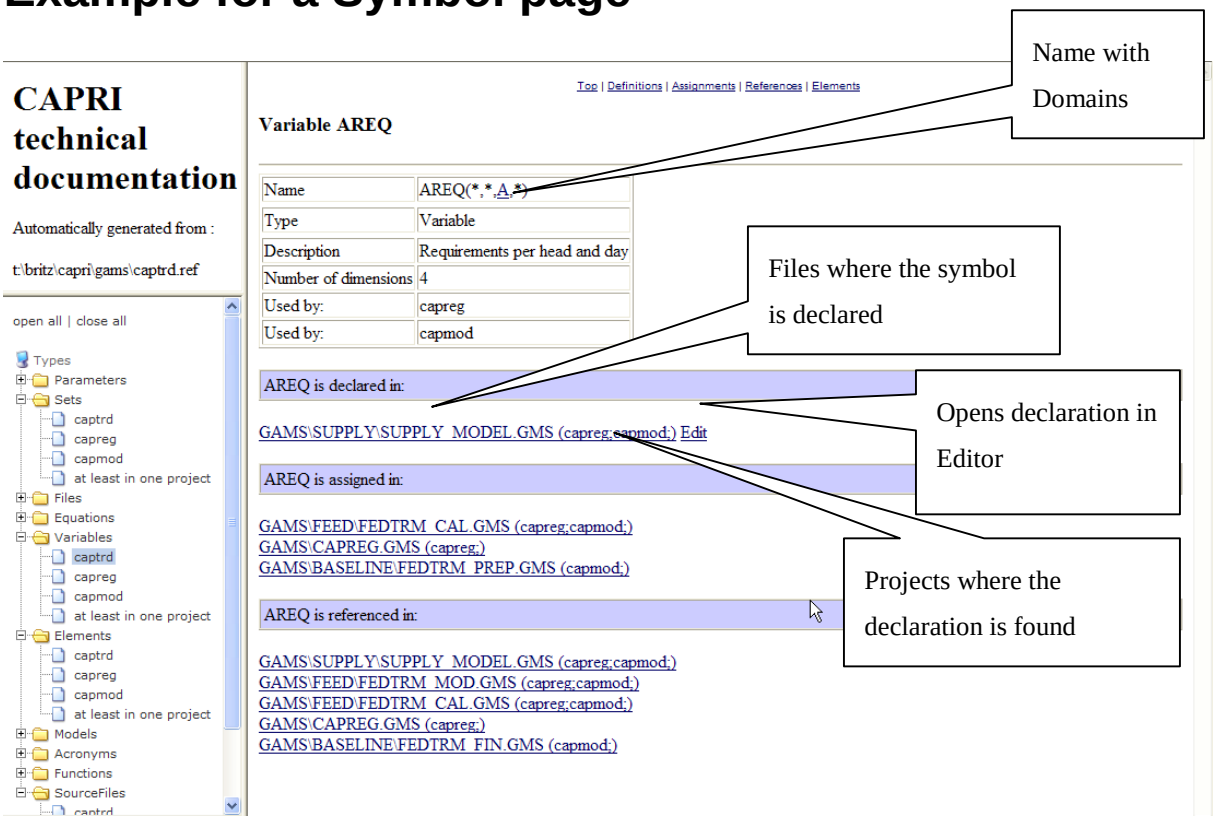
First implementation

An operational version is implemented as a JAVA application without a GUI. A zip file with the resulting HTML pages (open “index.ht , showing the results from generating simultaneously documentation f nd C

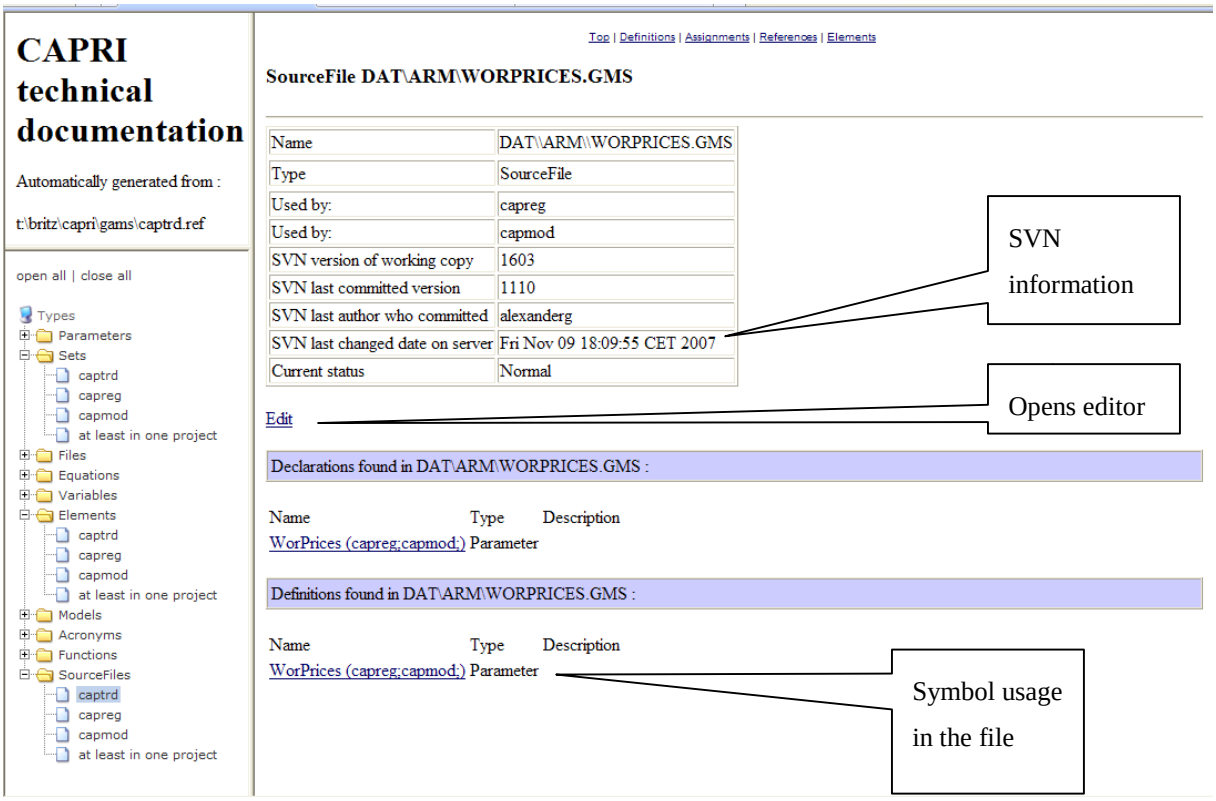
General overview



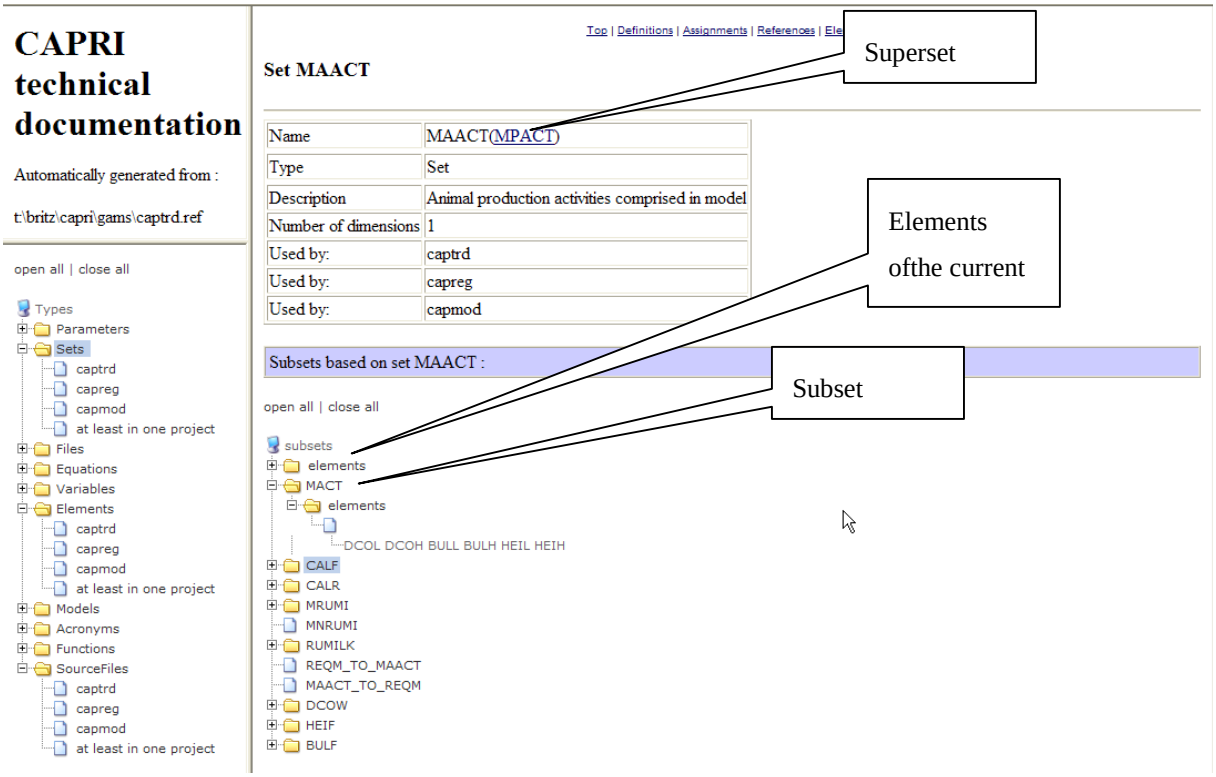
Example for a Symbol page



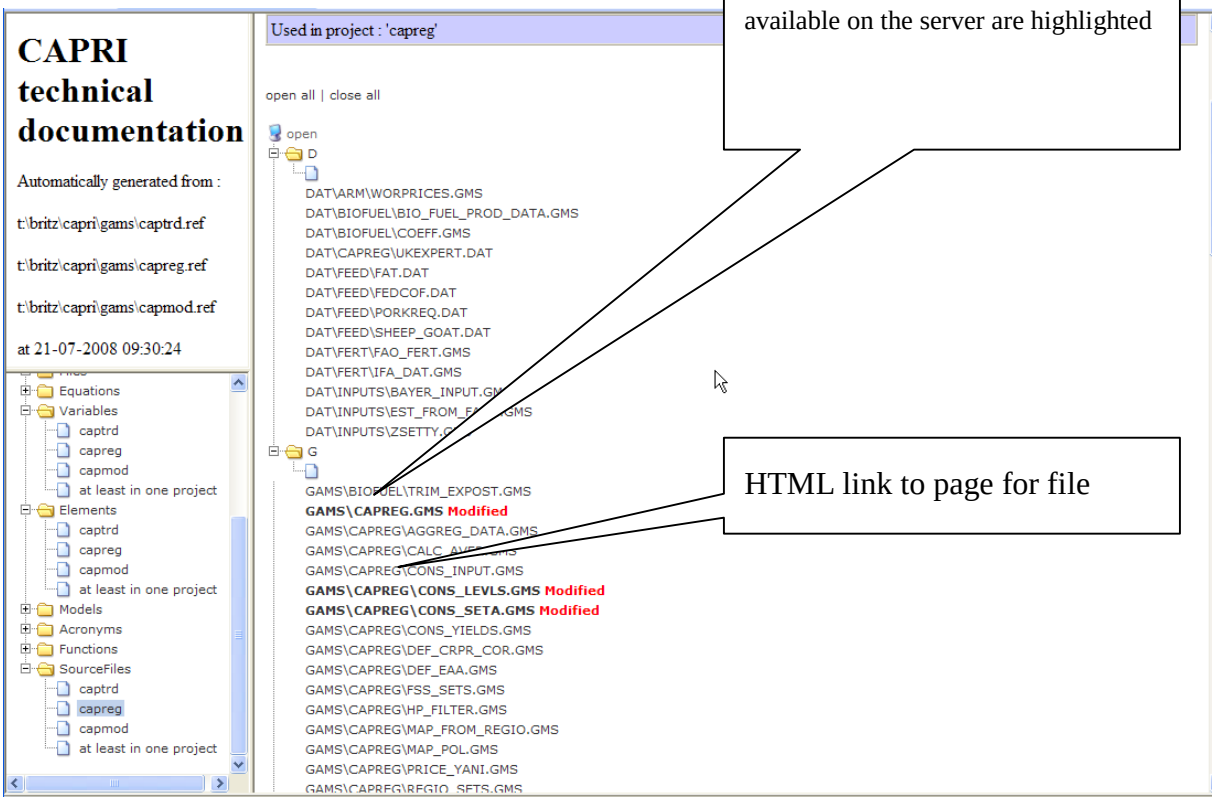
Example for a GamsSourceFile page



Example for a page for the a set



File list



Set element list