Synopsis User Manual

Synopsis User Manual

Table of Contents

List of Examples

Chapter 1. Introduction

1.1. What is Synopsis?

Synopsis is a tool for creating documentation from source code, using both comments and the code itself - the code is actually parsed rather than just searching for comments and particular expressions. Currently Synopsis can parse C++, Python and CORBA IDL, and generate documentation in a range of formats including HTML, DocBook, PDF and Dia graphs.

There are three stages in the documentation process: Parsing, Linking and Formatting. The data passed between each stage is an AST (Abstract Syntax Tree), a rich data structure which reflects the structure of the program. All three stages can be executed at once, or the AST can be stored to disk for later use.

Parsing is performed by language-specific parser modules. The most advanced is the C++ parser which does correct name resolution and generates extra information used to syntax highlight the source code and cross-reference symbols.

The Linking stage is where Synopsis derives most of its power, performing complex manipulations of the AST. At a minimum it handles the merging of multiple ASTs, eg: from different source files. It can manipulate the AST based on comments in the source, or perform things like renaming declarations to, for example, link directly to an IDL interface instead of a CORBA stub/skeleton.

The Formatting stage generates the documentation from the AST output by the Linking stage. There are a number of modules, but the most common is the HTML module. The HTML formatter is very flexible, itself consisting of multiple "Page" modules each of which generates a part of the documentation (the index, the list of modules, the class documentation, the inheritance tree, etc.)

Synopsis was written by Stephen Davies (aka Chalky) and Stefan Seefeld.

This manual was written by Stephen Davies.

1.2. How to use this manual

Synopsis is very configurable, and this manual attempts to document all the features that can be enabled and how to configure them properly.

The tutorial chapter shows some quick examples of how to get Synopsis working, and you can always check out the demos in the source distribution to see how some features work. Further chapters list all of the Parser, Linker and Formatter modules and all of the features/options available for each. Each feature has a detailed description, but for a quick reference you can also see the Config section of the Reference Manual generated from the Synopsis source code. You will also need the Reference Manual if you intend to write any extensions to Synopsis.

1.3. How to use Synopsis

Synopsis is a command-line tool, however a GUI may be included in future releases. Due to the highly configurable nature of Synopsis, it is not feasible to make a command line switch for each option. In order to aide testing and debugging of your configuration however, some of the more common options do have command line switches.

So, the main way to configure Synopsis is with a config file, which is actually a Python script (usually config.py). This script is executed in a sandbox by Synopsis, and then its namespace is inspected to extract the configuration options. If you already know Python then you will have no problems reading and writing the script, and if you don't, well Python is really easy to learn - the Python tutorial included with Python will get you up to speed in half an hour or less! You don't need to know any of the builtin functions or libraries to write a config script.

Synopsis uses three stages to generate documentation: parsing each source file to an AST (Abstract Syntax Tree), linking ASTs together and performing manipulations, and formatting the final AST. There are two ways to control this process:

1. Use a Makefile, and run Synopsis multiple times. This allows you to integrate Synopsis with your existing build system. A simple Makefile has three rules: one to parse each source file to a .syn file, another to link all the .syn files into a single .syn file, and a third to format the final .syn file. All the configuration is stored in the config.py file and the only switches given to Synopsis are to indicate the stage of processing and the input/output filenames/directories.
2. Use a Synopsis Project file. Project files are a new paradigm invented for the upcoming GUI which allows the entire process to execute with one synopsis command (eg: synopsis -P project.synopsis). It has some rules similar to make, with the exception that it does proper dependency checking and has flexible rules for selecting input files. The only downside is that in order to make the file easily readable and writable by the GUI it is much less readable and writable by hand.

This manual will concentrate on the Makefile and config.py method, as this is the simplest to learn. All of the configuration options are valid in both files, with the exception of those for controlling the processing of the three stages in the second method.

Synopsis can also integrate your project's documentation across multiple separate libraries or modules. An example is the best way to illustrate this: The Fresco Project has a Reference Manual for its IDL interfaces and its many implementation libraries and modules available online. Your project is a module which integrates with Fresco's display server, using both the IDL API and the display server's internal API. By downloading a .toc file (Table Of Contents) from the Fresco website, the documentation you generate for your project can link directly to Fresco's online documentation. For example, you have a method change_picture(Graphic g) where Graphic is an interface from the IDL API. In your documentation, Graphic will be a hyperlink to the documentation of the Graphic interface at the Fresco website. Neat huh? The Fresco documentation itself uses this feature to generate the documentation for each library separately, using .toc files to link to documentation of the libraries each library uses. See the HTML formatter section for more information.

1.4. How to format your comments

Synopsis is designed to be flexible in how you comment your code. All three stages of Synopsis deal with comments: the parser extracts them, the linker processes them, and the formatter formats them.

1.4.1. The parser

The parser is where the comments are extracted, and is the least flexible because the parsers use existing parser components (Python, IDL) or are hard coded in C++. Each parser looks for comments before declarations, and attaches them to the following declaration. The general rule is that comments must be immediately before the declaration, and there must be no blank lines between the comments and the start of the declaration. See below for an exception to this rule for C++.

The following is okay:

```
// A comment
/* Another comment
 * more text
 */
// Another comment
void the_declaration();
```

The following are not okay; the comments will be ignored:

```
// A comment

void the_declaration();

void the_declaration()
// A comment
{
  some_code();
}
```

The exception for C++ is when the option ’extract_tails’ (-Wp,-t on the command line) is used. In this case, all comments from the originating source file are included, but the ones which would not usually be included are flagged as ’suspect’. Comments that occurred after all declarations in a block (file, class, or namespace) are attached to a ’dummy’ declaration, which must be removed in the Linker stage using one of the ’dummy’ or ’prev’ comment processors. The former removes the dummy declarations and suspect comments, the latter does the same but checks for special syntax first, as shown:

```
int i; //< A trailing comment
enum Foo {
  Foo_A, //< A trailing comment for A
  Foo_B  //< A trailing comment for B attached to a dummy declaration
};//< A trailing comment for Foo
```

```
void func()
{
}
//< A trailing comment for func attached to a dummy declaration (if func was at the end
```

The ’<’ character at the start of the comment (after removing the prefix, // in this case) tells the ’prev’ comment processor to move the comment to the previous declaration. As you can see, this also applies to enumerators. The comment can be anywhere after the declaration - on the same line, the next line, or even further down.

1.4.2. The linker

The Linker deals with comments in the Comments linker operation. The Comments operation uses the config option ’comment_processors’ to find a list of Comment Processors to use, one after the other, on the whole AST. Each processor typically traverses the whole AST looking for comments, and performing operations on them.

The first comment processor in the list should always be one which strips away the comment characters, such as "//", "//.", "/* .. */", "//!", etc. After that, there are a number of processors you can use, for things like extracting JavaDoc style tags (@return, @param, etc), dealing with suspect comments and dummy declarations for C++ code, and deciding on a summary for the comment. The processor ’group’ is a processor that uses comments to organise declarations into ’groups’, within a class or namespace.

The comment processors are all written in Python, and you can specify your own in a config file to perform custom processing, or to deal with unusual comment styles.

See the chapter on the Linker for more information on this.

1.4.3. The formatter

Formatting comments is mostly concerned with dealing with the styling (which uses CSS in HTML, so you can change it from the stylesheet withing writing any code), and processing the tags found in the comments. The tags are already extracted by the linker (if you use the ’javatags’ comment processor), so it’s just a matter of formatting them in HTML or whatever. The only complexity is the @see tag, which has to try and find the referenced declaration to make a URL link.

See the chapter on the formatter you want to use for more information on comments (if there is any special support).

1.5. Where to get more help

Website: http://synopsis.sf.net

Task/Bug/Patch/File manager at SourceForge: http://sourceforge.net/projects/synopsis/

Mailing List: http://sourceforge.net/mail/?group_id=3169

Reference Manual: Your distribution might have a -doc package for this, else it is in the docs/RefManual directory of the source distribution. This is a generated source code reference for Synopsis, but also included is complete documentation for the Configuration files (see the Config link in the top right of any page).

Chapter 2. Tutorial

2.1. Running Synopsis

Several demos are included with the source distribution, but it is best to install Synopsis before trying them since they need the share data (icons, stylesheets etc) to work properly. There are demos for each parser, and some other feature demos, such as the Boost demo which documents Boost.Python and the Perceps demo which shows how to extend Synopsis to deal with "strange" commenting rules.

2.2. The easy way: copy a demo config + modify

The easiest way to get started using Synopsis is to copy a config file and maybe a Makefile from one of the demos.

Let’s say your project is called Player (it’s a CD player!) has the following layout:

```
/src/Player/Makefile
/src/Player/src/player.cc     # main file
/src/Player/src/gui.cc        # GUI code
/src/Player/src/driver.cc     # CD playing code
/src/Player/include/driver.hh # Classes for the CD playing code
/src/Player/include/gui.hh    # Classes for the GUI code
/src/Player/syn/              # Intermediate Synopsis files will go here
/src/Player/doc/html/         # Synopsis documentation will go here
```

Since it is a C++ project, we will copy the simple-config.py from the C++ demo. The C++ demo has several examples it compiles, so the config.py file there is a bit more complex than we need.

2.2.1. The Makefile

Your Makefile probably already has a variable FILES which lists the .cc files in src/, but Synopsis cares most about the header files. You will also need variables which contain the names of the .syn files generated by Synopsis’ parser, so add these lines to your Makefile:

```
# Project files
FILES = src/player.cc src/gui.cc src/driver.cc
INCLUDES = include/driver.hh include/gui.hh

# List of output .syn files for each input file
SYN_FILES = $(patsubst %,syn/%.syn,$(FILES) $(INCLUDES))
```

The patsubst line is a feature of (GNU) make that replaces a word (the %) with the word prefixed with syn/ (so the file goes in the syn directory) and suffixed with .syn (to mark it as a Synopsis file). This is done for both the source files and the include files.

The first step is to add a rule which will compile your source files (including .hh files!) into Synopsis .syn files containing the parsed AST:

```
# A pattern rule to parse each input file
$(SYN_FILES): syn/%.syn: %
        # Here $@ is the output .syn file and $< is the .cc or .hh file
        synopsis -c config.py -Wc,parser=C++ -o $@ $<
```

The first line there is a pattern rule. It says that for each of the SYN_FILES, the output syn/%.syn depends on %. This means, for example, that syn/src/player.cc.syn depends on src/player.cc.

The arguments to synopsis are:

-c config.py

  Use the config file "config.py". This is the Python script with the configuration options in it. We will get to this file later...

-Wc,parser=C++

  Gives the "parser=C++" option to the configuration module. Note that the -Wx,foo,bar style of command line option is borrowed from the GCC parser. Options are separated by commas and are passed to the relevant modules. Other modules you can address in this way are the parser ("-Wp,-m,-s,myfile.cc.links"), the linker ("-Wl,-s,Foo::Bar") and the formatter ("-Wf,-s,styleshee.css"). In this case we are telling Synopsis to use the parser config with the name C++. This also tells it that the inputs are source files and must be parsed. See the config.py section below for more on this.

-o $@

  Output the AST to $@, which will be expanded by Make to be the output file, eg: syn/src/player.cc.syn

$<

  Input file, which will be expanded by Make to be the first dependancy - the source file, eg: src/player.cc

Next is a rule to combine the multiple .syn files into a single one containing the whole AST for your project:

```
# The combined AST file which is the result of linking all files together
COMBINED = syn/combined.syn

# Make the combined file depend on all the intermediate files
$(COMBINED): $(SYN_FILES)
        # Link all the files together
        synopsis -c config.py -Wc,linker=Linker -o $(COMBINED) $(SYN_FILES)
```

The arguments to synopsis here are:

-Wc,linker=Linker

: Similar to the previous -Wc option, this one tells Synopsis to use the linker config named as "Linker" in the config file. Since we didn’t specify a parser, Synopsis assumes that the inputs are binary .syn files containing parsed ASTs.

$(SYN_FILES)

: A small note that make expands this variable to many files (5 in our example). Multiple AST files can be passed to the Linker, or to a formatter (in which case Synopsis links them with default options).

Note that although there is only one Linker module, you can have different configurations for it to do different things such as different commenting styles or stages of processing.

Finally, a rule to format the combined AST into the HTML output. This step and the previous one can be done in one go, but this way you can generate multiple output formats from the one AST.

```
# The (main) final output file
HTML = doc/html/index.html

# Make the output file depend on the combined file
$(HTML): $(COMBINED)
        # Run synopsis to generate output
        synopsis -c config.py -Wc,formatter=HTML $(COMBINED)
```

The arguments should be obvious by now:

-Wc,formatter=HTML

: Tells Synopsis to use the formatter config named "HTML".

Some housekeeping rules come last: A nice "doc" target so you can run "make doc", and a clean target so you can run "make clean". If you already have a clean target, just add this to your existing one.

```
# A doc target which creates the output file
doc:    $(HTML)

# A clean target which removes all the .syn files
clean:
        rm -rf $(COMBINED) $(SYN_FILES)
```

The final Makefile looks like this:

```
# Project files
FILES = src/player.cc src/gui.cc src/driver.cc
INCLUDES = include/driver.hh include/gui.hh

# List of output .syn files for each input file
SYN_FILES = $(patsubst %,syn/%.syn,$(FILES) $(INCLUDES))

# The (main) final output file
HTML = doc/html/index.html

# The combined AST file which is the result of linking all files together
COMBINED = syn/combined.syn

# A doc target which creates the output file
doc:    $(HTML)

# A clean target which removes all the .syn files
clean:
        rm -rf $(COMBINED) $(SYN_FILES)

# Make the output file depend on the combined file
$(HTML): $(COMBINED)
        # Run synopsis to generate output
        synopsis -c config.py -Wc,formatter=HTML $(COMBINED)

# Make the combined file depend on all the intermediate files
$(COMBINED): $(SYN_FILES)
        # Link all the files together
        synopsis -c config.py -Wc,linker=Linker -o $(COMBINED) $(SYN_FILES)

# A pattern rule to parse each input file
$(SYN_FILES): syn/%.syn: %
        # Here $@ is the output .syn file and $< is the .cc or .hh file
        synopsis -c config.py -Wc,parser=C++ -o $@ $<
```

You can find this Makefile in demo/Tutorial/Player.

2.2.2. The config.py file

Now that the Makefile is written it's time to look at the config file. The simple-config.py looks like this:

```
# Config file for C++ demos
# Only some files have comments in //. style

from Synopsis.Config import Base

class Config (Base):
    class Parser:
        class CXX (Base.Parser.CXX):
            main_file = 1
        modules = {
            'C++':CXX,
        }

    class Linker:
        class LinkerJava (Base.Linker.Linker):
            comment_processors = ['java', 'javatags', 'summary']
        modules = {
            'Linker':LinkerJava,
        }

    class Formatter:
        class HTML (Base.Formatter.HTML):
            stylesheet_file = '../html.css'

        modules = Base.Formatter.modules
        modules['HTML'] = HTML
```

It is important to remember that the config.py file is a real python script and is executed just like a Python program. This has the advantage of allowing familiar syntax and flexibility, but at the expense of a little security risk - however generating documentation for source code is probably less risky than running the source code itself, and a config.py file is easier to check.

The first Python line is to access the "default" config options in the module Synopsis.Config. The default options are stored in the Base class, which has many subclasses for the different modules. You can see these in either the Config.py file itself, or in the generated reference manual. It is not strictly necessary to use the Base class, but it simplifies the config file somewhat by providing many sensible defaults.

The rest of the file defines a class called Config, which has three subclasses called Parser, Linker and Formatter, representing the three stages of processing. Each of these subclasses contains a "modules" dictionary which lists different named configurations for that stage. Usually the only use for having different configurations is to use different modules (different parsers, different formatters), but you can also have multiple configurations for the same module. The most common example of this is to have multiple Linker configurations for different commenting styles and other options.

Let's examine the file a section at a time:

```
class Config (Base):
```

The Config class derives from the Base class (that we imported at the start of the file) which has a constructor to sort out the configurations in the config class. Remember that Python uses indentation to indicate scope, so the lines after this are all indented to indicate that they are part of the Config class.

```
class Parser:
```

This is the first subclass, containing configurations for the Parser stage.

```
class CXX (Base.Parser.CXX):
    main_file = 1
```

This is the only parser configuration in this config file. Configurations are classes too, but they are instantiated when they are used so they can have constructors to set options at run time. This configuration class is called "CXX" and derives from the class Base.Parser.CXX which has some default config options and also a constructor (so be sure to call it if you write your own).

Configuration options are set as attributes in the config object, which the parser examines when it runs. The only one here sets the "main_file" option to "1". See later chapters in this manual for more options, or the reference manual for a complete list.

```
modules = {
    'C++':CXX,
}
```

This little bit of code sets an attribute in the Parser class called modules, to a dictionary (note the braces aka curly brackets) of named configurations. Here 'C++' is the name of the configuration to be used on the command line, and CXX (no quotes!) refers to the CXX class in the current scope. Note that the class was called CXX rather than C++ since C++ is not a valid class name.

There is just one thing missing here: how does Synopsis actually know to use the C++ parser for this config? Surely it doesn't go by the name of the class... The answer lies in the fact that we derived from the default configuration class Base.Parser.CXX. Looking in the Config.py file shows (amongst other things):

```
class CXX:
    name = 'C++'
```

Synopsis appends this "name" attribute to "Synopsis.Parser." to make the module name "Synopsis.Parser.C++". This means that all parsers must be in the Synopsis.Parser package, for now.

To use a parser for a different language, the easiest thing to do is derive from one of Base.Parser.IDL or Base.Parser.Python, as appropriate. Note that the different parsers generally have different options.

Getting back to the config file, the Linker section is pretty similar to the Parser section, except for this line:

```
comment_processors = ['java', 'javatags', 'summary']
```

The syntax here is setting the comment_processors attribute to a list of strings. The (square) brackets indicate a list (parentheses aka round brackets indicate a tuple, which is an immutable list). Strings can be either single or double quotes - Python makes no distinction between the two so it’s just a matter of style.

This particular option tells the Linker to apply the three comment processors called ’java’, ’javatags’ and ’summary’ in that order. Comment processors are operations that use the comments attached to declarations to perform manipulations of either the comments or the AST itself:

java

Looks for comments with the java commenting style of /** foo bar */. Any comments not matching this format are removed, and then the /** and */ strings are also removed. Intermediate lines must begin with a *, so a multi-line comment looks like:

```
/** The first sentence is a summary.
 * The rest of the comment goes here, with each line
 * beginning with a star. The closing star-slash can go
 * either by itself or at the end of a line.
 */
```

javatags

Comments can have special "tags" which indicate either processing or formatting. This processor extracts the JavaDoc style tags from the comments and stores them with the comment for other processors or formatters to use. The tags must be at the end of the comment, but this allows them to span multiple lines if necessary. An example of JavaDoc tags is:

```
/** Returns the height of the given person.
 * @param firstname the first name of the person
 * @param surname the surname of the person
 * @return the height of the person or 0 if the person was not found
 */
float getHeight(std::string firstname, std::string surname);
```

summary

This processor extracts a summary of each comment and stores it with the comment, allowing the documentation to display a (usually one line) summary next to declarations in order to keep the pages tidy, and show the full comments elsewhere (e.g. further down the page). This processor also combines the comments for each declaration, so it should be the last one in the list.

See chapter XXX for a full list of comment processors.

The HTML formatter we will have to change slightly. The stylesheet_file option tells the HTML formatter where to read the stylesheet from, and since the demo’s are designed to run without Synopsis being installed, this is probably wrong. The default is to use the one installed in $prefix/share/Synopsis, so you can either delete the line or set it to the correct path. If you delete it, remember that Python needs the word "pass" for an empty class (or function):

```
class HTML (Base.Formatter.HTML):
    pass
```

Finally, the way the modules dictionary is set for the Formatter is a bit different. Here we first set it to the default value from the base class, then overwrite the entry for HTML with the new HTML formatter configuration. By using the default value like this, you get (default) configurations for all the different formatters.

2.2.3. The output explained

That’s it! Just run "make doc" and hopefully it should all work. You will see Synopsis being run for each input file, to combine the ASTs, and the format the output.

The HTML is output to whatever directory you set in the Makefile, which was written above as doc/html/. Point your web browser at the index.html file there to view the documentation, and you will see three frames (using the default configuration) similar to JavaDoc documentation:

Top-left "contents" frame

> This frame shows a tree heirarchy of the modules or files in the project. Clicking on a link opens that module or file in the "index" frame below.

Bottom-left "index" frame

> This frame shows an index of the currently selected module or file, but only shows child classes, structs, namespaces or modules. When a module page is loaded in this frame and you have JavaScript enabled, a more detailed page will open in the main content area. For file pages you will need to click the "File Details" link.

Right "main" frame

> This frame shows the main documentation. At the top of each page you can see a list of documentation areas you can visit: The Modules and Files open in the top-left frame, the Inheritance Graph/Tree and Name Index load in the main frame. For "scope pages" (i.e. the pages for classes and namespaces, including the global namespace which is the default page) the page is divided into summary and detail sections: the summary shows all the declarations in this class or namespace, with a summary for each. For declarations with more comments than just the summary, the name of the declaration will be highlighted, and clicking on it will take you to its detailed information further down the page.

You can also try out the other formatters. To see which formatters are available, run "synopsis -l". To use a particular formatter, type:

```
$ synopsis -c config.py -Wc,formatter=Dia -o player.dia syn/combined.syn
```

to use configuration from the config.py file (in our config file we just inherited the default options anyway), or you can just use the default options by typing:

```
$ synopsis -f Dia -o player.dia syn/combined.syn
```

You can also pass command line options to the formatter with the following syntax:

```
$ synopsis -f Dia -Wf,-p -o player.dia syn/combined.syn
```

This passes the -p option to the formatter. For the Dia formatter this tells it not to include parameters.

2.3. The hard way: create from scratch

An alternative to copying from a demo is to create your Makefile and config.py file from scratch. You have the freedom to generate the Makefile and/or config.py file from a configure script if you want - for an example of this see the Fresco project.

TODO: Write tutorial. See the earlier tutorial in the mean time.

2.4. The powerful way: new project file format

The Project file provides a more powerful, more integrated method of driving Synopsis than a Makefile. However, it is harder to work with. The intended usage is from a GUI, however a GUI is still forthcoming (see the next section).

The project file is still a Python script, but does away with the methods and nested classes, instead being one class called Project which has a number of attributes. Some of these attributes are quite complex, but are stored in a readable ’repr’ type format. Configuration for modules is stored as an object instance rather than as a class, and is represented in the file as an object instantiation, like so:

Example 2-1. How object instantiation works

```
# The ’struct’ class is included at the top of the config file. It
# basically stores attributes given to the constructor
class struct:
  def __init__(self, **args):
    for key, value in args.items:
      setattr(self, key, value)

class Project:
  # Instantiate an object:
  my_object = struct(name = "foo", magic_number = 42)

  # And another one, this time formatted nicely:
  some_object = struct(
    name = "hi",
    some_list = [1, 2, 3],
    # Here an attribute is a nested object:
    nested_object = struct(
      type = "foo",
      fred = "baz"
    )
  )
```

The new Project structure defines not only configuration for the individual modules, but the relationships between different stages of the document generation process. The process is split into "Actions", each of which generally has inputs and outputs, with some exceptions:

SourceAction

Selects the source files to use in the project. It has no input, and its output is just the names of the individual files. The SourceAction has a number of rules which it follows in turn to decide which files to use. The rules are set by the configuration, and can select simple file names, use recursive searches and glob expressions, or exclude previously selected files. The ability to exclude certain files is useful for avoiding the appearance of temporary, backup, or cvs-related files in the documentation. See the next section for details on the available rules.

ParserAction

Takes a list of files from one or more SourceActions at input, and applies a Parser to each one in turn. The output is a number of AST's, one for each input file. The ParserAction contains a config object with options for the parser.

CacherAction

Can take as input any number of ASTs (from eg: ParserAction, LinkerAction). Outputs the same ASTs, but stores the ASTs on disk so that they don't need to be regenerated each time if the timestamps have not changed. For C++ files all dependencies (#included files) will have their timestamps checked. For other languages only the actual source files will be checked.

LinkerAction

Can take one or more ASTs as input, and outputs a single AST. The name comes from the process of linking a number of ASTs, typically from different source files, into a single cohesive AST. The Linker is more powerful than that however, allowing complex manipulations of the AST. These manipulations can be based on configuration only, or rely on comments extracted from the source files. The LinkerAction contains a config object to contain all the linker options available.

FormatAction

The FormatAction is the tail end of the chain, and takes a single AST which it converts into some output format. A number of output formats are supported, with varying degrees of completeness and usefulness. The HTML formatter is the most developed, having an immense range of configuration options.

The configuration options for the config objects in the Parser, Linker and Formatter objects are described later in this manual, and are the same as the old config format just with a different syntax.

2.4.1. Project class

```
class Project:
    # attributes go here. Must be indented!
    name = 'project name'       # string - name of project
    verbose = 0                 # boolean
    data_dir = './'             # string - path (currently unused)
    default_formatter = 'name'  # string - the name of the default formatter to
                                # use if none is specified on the command line.
    channels = [ <list of channels> ]
    actions = [ <list of actions> ]
```

The main element of the config file is the "class Project". It has a number of attributes, including a string name, verbose flag, a list of actions, a list of channels, and a default formatter to use if none is specified. The channels are the connections between actions, and are simply tuple pairs of names of actions, e.g.:

```
    channels = [
        ('C++ Parser', 'File Cacher'),
        ('File Cacher', 'Linker')
    ]
```

Be careful that the action names in the channels list match the names of the actual Actions in the actions list.

2.4.2. Action object

An action object, part of the list of actions in Project, is a list of attributes (not a struct!). The first 4 attributes are always the same, but different Actions have different uses for the remaining attributes (remember, a list has variable length). The following example shows all action types:

```
    actions = [
        # string type, X/Y coords, string name, extra attributes...
        ['SourceAction', 100, 100, 'Boost Sources',  [ <list of SourceAction Rules> ] ],
        ['ParserAction', 200, 100, 'C++ Parser',     struct(key=val, key=val, ...) ],
        ['CacherAction', 300, 100, 'File Cacher',    'directory' or None, 'file' or None
        ['LinkerAction', 400, 100, 'Linker',         struct(key=val, key=val, ...) ],
        ['FormatAction', 500, 100, 'HTML Formatter', struct(key=val, key=val, ...) ]
    ]
```

As you can see, all types use the same format for the first four attributes. The first is the type of action. The 2nd and 3rd are the X and Y coordinates to use to display the action in the GUI. The 4th is the name of the Action.

For SourceActions the 5th attribute is a list of SourceAction Rules (see the next section).

For CacherActions the 5th and 6th attributes are directory name and file name respectively. One and only one of these must be set (the other must be None). If directory is set, then the inputs to the action will be cached in the given directory and the outputs will be the same as the inputs. If the file is set, then the action will have no inputs, and one output which is loaded from the given file. Note that this second mode of operation is not really caching but more like loading, and can be used to load an AST from a file which you don’t have or want to use source code for.

For all other Action types the 5th attribute is a Config object. For information on these see the documentation for the relevant chapters of this manual (Parsers, Linker and Formatters).

2.4.3. SourceAction Rules

The rules used to select files in the SourceAction are stored in a list, where each rule is a tuple. The first element of the tuple is a string denoting the type of rule, and the rest depends on the type. Available types of SourceAction rules are:

Simple

A ’simple’ rule that contains a list of filenames as the second tuple element. The filenames may be absolute or relative. For backwards compatibility, a single filename may be used instead of a list.

```
(’Simple’, [’fred.py’, ’baz.py’])
```

Glob

Allows easy selection of a number of files in a project or subdirectory, without having to update the config when extra files are added or removed. There are three extra tuple elements: a list of directories, a glob string, and an integer recursion flag. The directory names are just strings, and may be absolute or relative. The glob string is a glob expression as used by unix shells: the special characters are ’*’ which matches to 0 or more characters, and ? which matches any one character. The recursion flag can be 0 or 1. If 1, then subdirectories of the listed directories are also searched.

```
(’Glob’, [’src/’, ’tools/’], ’*.cc’, 1)
```

Exclude

Allows files matched by previous rules to be excluded from the list. Remember that the rules are processed in order, so later rules can add more files again. There is only one extra field which is a glob expression to match the whole pathname. In order to make matching subdirectories easier, the filenames are always considered to have a ’/’ prepended before comparing to the glob expression. This means that if a CVS directory is in the current directory and its files’ relative pathnames resemble ’CVS/Root’, you can still match it with ’/CVS/’.

```
(’Exclude’, ’/CVS/’)
```

Dir

Backwards compatibility for an old version of non-recursive Glob: Has a single directory and a glob expression.

```
('Dir', 'src/', '*.cc')
```

Base

Backwards compatibility for an old version of recursive Glob: Has a single directory and a glob expression.

```
('Base', 'src/', '*.cc')
```

All rules are processed in the order that they appear in the list. The effect of a rule is to add remove files to/from a list. Therefore to explicitly exclude a file, the exclude rule should be at the end.

Note that this is the opposite to permission type rules you might be used to, since in this case we don't know what files there are until a rule is processed. With permissions you know what user/file you are checking permissions for so it's usually the first matching rule which is used - hence exclusions are usually placed first for permissions.

Another special case is when you are using the 'multiple_files' feature of the C++ parser - in this case the first rule must be a Simple rule with the main filename (e.g.: Python.h). This is documented at the end of the C++ Parser section.

2.4.4. Command Line Usage

To use the project file and generate output for the default formatter (specified in the config), use a command line like so:

```
$ synopsis -P project.synopsis
```

You can specify a formatter to use other than the default formatter like so:

```
$ synopsis -P project.synopsis -Wc,formatter=MyFormatter
```

Synopsis will ensure that all Actions connected to the formatter are also executed.

See the Boost demo for an example of how to use the project file format.

2.5. The future way: use the GUI

In some future release there will be a GUI to allow easy configuration of Synopsis.

You can preview the GUI by running bin/synopsis-qt in the source distribution, but be warned: it is pre-alpha, and not all features are supported yet, meaning you still have to edit the project file by hand.

Chapter 3. The Parsers

3.1. C++ Parser

The C++ parser reads C++ source code and generates an AST. It is the most advanced Synopsis parser, using the following components:

- A modified version of UCPP by Thomas Pornin. The modifications are to improve C++ standards conformance, and add features needed by Synopsis such as storing information about #includes and #defines, and recording the position of every macro expansion.
- A modified version of OpenC++ (OCC) by Shigeru Chiba and others. OCC is used to generate a "parse-tree" from the preprocessed source code. The parse-tree is a data structure that directly represents the syntax of the source code. OCC uses Hans Boehm's garbage collector, but this can be disabled by passing --disable-gc to configure on platforms like Cygwin where it causes problems.
- Code to "walk" the parse-tree and build an Abstract Syntax Tree. A symbol table is also maintained which is used to correctly resolve all types according to the C++ standard.
- Code to record the positions of all macro expansions, keywords, comments, type and name references if SXR or XREF (see below) is enabled.
- Miscellenous code to convert the C++ AST to a Python data structure, maintain the list of files, implement a Python module etc.

Like a compiler, the C++ parser has to parse the whole translation unit for each file to make sure it has the correct type references. Unfortunately this means it has to wade through all the standard library headers for each input file which, like with a compiler without precompiled headers, can take a long time. Luckily the C++ parser is written in C++ for speed and has been optimised in critical areas. If you are not using SXR or XREF then it skips all method bodies which speeds things up, and even if you are it only has to resolve type and function call information - there is no code generation or optimisation going on.

Fortunately you can parse multiple headers in one go if you have a main header file which includes everything else (such as how python2.2/Python.h includes the other 41 Python header files), and if you are using a Project config file. See the multiple_files option below.

3.1.1. SXR and XREF

So what are SXR and XREF?

SXR stands for Synopsis Cross Reference. It is a combined feature of the C++ parser and the HTML formatter which displays the source code but with hyperlinks for every type reference and every name reference.

This means that if you click on a method signature in the source code, you will be taken to the documentation. Each declaration in the documentation will also have a link to the source code. Variable and type references in the source code also have "titles", which show as tooltips if you hover the mouse over them, telling you what they are (eg: "parameter Person" or "member variable std::string"). Note that because Synopsis uses a real C++ parser the links are always to the right declarations - unlike other systems which use purely string based references which are often ambiguous!

XREF stands for Cross Reference. This feature creates an index of all references to every declaration, including function and method calls. This allows you to follow a call graph, or see where a method or class is used (useful for refactoring!)

Both of these features need to be coordinated with both the C++ parser and the HTML formatter.

For SXR you need: C++ Parser: syntax_prefix. HTML Formatter: The pages option must include ’FileSource’.

For XREF you need: C++ Parser: xref_prefix. Linker: The operations list must include ’XRefCompiler’, and XRefCompiler must be configured. HTML Formatter: The pages option must include ’XRefPages’, and XRefPages must be configured.

See the respective sections for details of these configuration options.

Note that in the current release there are some limitations with respect to expressions that are understood, and template instantiations are not supported very well. This means that in the source code some methods will not be hyperlinked and some cross references will be missing.

3.1.2. Options

name (string)

    Name of this config object: Must be ’C++’

verbose (boolean)

    Verbosity flag. For config files, this attribute is set by the constructor, but only if ’verbose’ was passed as a config option.

    Can be set to true from the command line with "-Wp,-v"

main_file (boolean)

    Flag that selects whether Synopsis should only store the AST generated from the file being processed, and not from any #included files. If you are using the multiple_files option then the AST generated from each of the named files is stored, but not from any other #included files.

The default is 1 (true).

Can be set to true from the command line with "-Wp,-m"

basename (string)

A file path to strip from the start of all filenames before storing. Setting this option for example will remove any redundant parent directories in the HTML FileListing page.

Can be set from the command line with, e.g.: "-Wp,-b,../some/basename/"

include_path (list of strings)

A list of paths to add to the include path. For example: ['/usr/local/corba/', '../include']

Can be set from the command line with, e.g.: "-Wp,-I,/usr/local/corba/,-I,../include"

defines (list of strings)

A list of defines to pass to the preprocessor. For example: ['FOO', 'BAR=true']

Can be set from the command line with, e.g.: "-Wp,-D,FOO,-D,BAR=true"

preprocessor (string)

Which preprocessor to use. Not setting this causes the builtin ucpp to be used, which can track macro expansions when doing SXR stuff and extract macro definitions for the documentation. Setting it to 'gcc' will cause gcc (well, really g++) to be used instead, for use only in cases when ucpp can't parse your standard libraries (usually because of compiler specific syntax). There are no other settings.

Can be set to gcc from the command line with "-Wp,-g"

extract_tails (boolean)

If set to true, then the parser will look for trailing comments before close braces. If it finds them, it will create dummy declarations to hold the comments. If you set this, you should probably also use the 'dummy' or 'prev' comment processors in the Linker options otherwise you will see the Dummy declarations in your documentation.

Can be set to true from the command line with "-Wp,-t"

storage (string)

Deprecated, see syntax_file which is the same thing.

syntax_prefix (string)

If set, must be a string which defines a prefix to store syntax info into. The source filenames are appended to the prefix to create the output filename, so it should be a directory name to avoid clashes (there is no suffix!). For example, if your file is "src/foo.cc" and prefix is "syn/" then the syntax information will be stored in "syn/src/foo.cc".

xref_prefix (string)

If set, must be a string which defines a prefix to store xref info into. See syntax_prefix.

syntax_file (string)

If set, must be a string with the file to store syntax info into. Setting this also causes the parser to look more carefully at the input file, so that it can generate the links from inside blocks of code (otherwise it skips over them). Note that the syntax info can only hold syntax information about one source file, so this option is of limited use in a config file unless you only have one file.

Can be set from the command line with, e.g: "-Wp,-s,syn/". You might use it in your Makefile as "-Wp,-s,$<.links" ($< is the first dependency, usually the source file being parsed).

xref_file (string)

If set, must be a string with the file to store xref info into. Setting this also causes the parser to look more carefully at the input file, so that it can generate the links from inside blocks of code (otherwise it skips over them). Note that the xref info can only hold xref information about one source file, so this option is of limited use in a config file unless you only have one file.

Can be set from the command line with, e.g.: "-Wp,-x,xref/". You might use it in your Makefile as "-Wp,-x,$<.xref" ($< is the first dependency, usually the source file being parsed).

fake_std (boolean)

If set, this causes the parser to construct a fake using directive from the std namespace to the global namespace. In effect, this fixes problems seen with using the stdc++ headers for gcc 2.95.x where most things dont get placed in the std namespace.

Can be set to true from the command line with "-Wp,-f"

multiple_files (boolean)

This option can only be used with the Project file format. If set to true then the parser accepts a list of files, and stores the generated AST for all of them in the one output. Only one file is parsed, so the others must be included by the main file (or by one of the files it includes, etc.). If syntax_prefix and/or xref_prefix is set then the extra files will get syntax and/or xref info recorded into the appropriate files. Only one AST is output, but it is as if the ASTs for the individual files were already linked. That, and the fact that system headers will only be parsed once, can lead to huge time savings!

To use this option, your Project file must have a single SourceAction connected to this ParserAction. The SourceAction should have a Simple rule first which is the main sourcefile, and any number of other rules to select other files to record the AST for.

3.2. Python Parser

The Python parser uses Python's parser library to create an AST from Python source code. This is adequate for the purposes of extracting classes, methods and their comments, but prevents the advanced features like syntax highlighting since the parse tree does not store line number information. There is no support for SXR or XREF features.

3.2.1. Options

The following options are available.

name (string)

Name of this config object: Must be 'Python'

verbose (flag)

Verbosity flag. For config files, this attribute is set by the constructor, but only if 'verbose' was passed as a config option.

Can be set from the command line with "-Wp,-v"

basename (string)

A file path to strip from the start of all filenames before storing. Setting this option for example will remove any redundant parent directories in the HTML FileListing page.

Can be set from the command line with, e.g.: "-Wp,-b,../some/basename/"

3.3. CORBA IDL Parser

The CORBA IDL Parser uses OmniORB's (Python) IDL compiler library to parse IDL files and generate an AST. You will need to have the omniidl package installed to use this parser. There is no support for SXR or XREF features.

name (string)

Name of this config object: Must be ’IDL’

verbose (boolean)

Verbosity flag. For config files, this attribute is set by the constructor, but only if ’verbose’ was passed as a config option.

Can be set to true from the command line with "-Wp,-v"

main_file (boolean)

Flag that selects if should only store the AST generated from the file(s) being processed, and not included files. The default is 1 (true).

Can be set to true from the command line with "-Wp,-m"

basename (string)

A file path to strip from the start of all filenames before storing. Setting this option for example will remove any redundant parent directories in the HTML FileListing page.

Can be set from the command line with, e.g.: "-Wp,-b,../some/basename/"

include_path (list of strings)

A list of paths to add to the include path. For example: [’/usr/local/corba/’, ’../idl’]

Can be set from the command line with, e.g.: "-Wp,-I,/usr/local/corba/,-I,../idl"

keep_comments (boolean)

If true, comments will be stored in the AST. The default is true.

Can be set to true from the command line with "-Wp,-K" (note upper case K)

Can be set to false from the command line with "-Wp,-k"

Chapter 4. The Linker

The Linker is one of the more powerful features of Synopsis. It is actually a slight misnomer, since it does more than just linking multiple ASTs together - it can also move subtrees around, remove subtrees, map declaration names or type names, and perform various operations to do with comments.

Unlike the Parsers and Formatters, there is only one Linker. It does however have a number of modules to perform various operations on the AST. The operation modules to use are specified in the "operations" config option. They should generally be in the order given by the following sections. See below for a list of all the Linker options.

4.1. Unduplicator

The unduplicator is responsible for cleaning up the mess from merging multiple ASTs. The problem is that each AST has its own copy of any duplicate classes and functions, and many types (e.g.: parameter types, return types, variable types) will be pointing to the wrong copy. The unduplicator walks the whole AST ensuring that each declaration only occurs once, that the types dictionary refers to the one correct declaration, and that every type refers to the correct declaration. It takes special care to handle "Modules" (eg: C++ namespace and IDL modules) by combining them into MetaModule AST nodes. Nested classes declared outside the original class are also taken care of.

There are no options for the Unduplicator, but you will always need it if you are combining multiple ASTs.

4.2. Stripper

The stripper compares every name in the AST to a list of prefixes. Note that names here are scoped names, such as "boost::any::any_base::get_obj()" or "Synopsis.Formatter.HTML.core.format()". If a name matches, the prefix is stripped. If a name doesn't match any prefix, it is removed from the AST. If no prefixes are specified then the Stripper does not do anything (it doesn't even walk the tree so there is no performance penalty).

This can be used to select parts of an AST to generate separate documentation. For example, the Synopsis RefManual splits the whole Synopsis AST into sections for formatting into different .info files.

4.3. NameMapper

This simple operation adds a prefix to every declaration and type name in the AST. The Synopsis RefManual uses this to move the whole C++ AST into the Synopsis.Parser.C++ module.

4.4. Comments

This is the most complex Linker operation, since it is itself composed of multiple "comment processors" which work on comments attached to declarations. They should generally be in the following order, if present. You will always need a comment-identifier processor first, and the summary processor last.

4.4.1. ssd

This processor identifies comments in the //. style. Any comment line not starting with //. will be removed, and the //. is stripped from those that do have it.

```
//. This is a comment
//. With multiple lines
```

4.4.2. ss

This processor identifies comments in the // style. Any comment line not starting with // will be removed, and the // is stripped from those that do have it.

```
// This is a comment
// With multiple lines
```

4.4.3. java

This processor identifies comments in the /** .. */ style. Any comment not beginning with /** and ending with */ will be removed, and the /**, */ and any * at the start of a line are stripped from those that have them.

```
/** This is a comment
 * With multiple lines */
```

```
/** This is a comment
 * With multiple lines
 */
```

4.4.4. qt

This processor identifies comments in the Qt style. There are two types of comments: brief and detail. Brief starts with //! and there must be only one. Detail comments are surrounded by /*! and */.

```
//! Summary of this comment
/*! More info about the declaration
    There can be multiple lines.
    Must end with */
```

4.4.5. group

Looks for sequences of declarations to group together by looking for a ’{’ in a comment and a ’}’ in a following comment in the same scope. The ’{’ and ’}’ must be at the beginning of a line, but may have a group name after them on the same line. For example:

```
class Foo {
public:
    //. Methods for accessing attributes of
    //. this Foo
    //. { Accessor methods
    //. Get a Bar
    Bar* get_bar();
    //. Set the Bar
    //. @param bar the Bar to set
    void set_bar(Bar* bar);
    //. }
};
```

Note that you will need to set ’extract_tails’ in the C++ parser for the above example to work, since the last comment is between the last declaration and a closing brace! If you set extract_tails, you will also need one of ’dummy’ or ’prev’.

4.4.6. dummy

Removes the dummy declarations created by setting "extract_tails" in the C++ parser. Any comments they had are lost. See the ’prev’ processor below.

4.4.7. prev

Checks comments to see if they start with a ’<’. If so, they are moved to the previous declaration. Any dummy declarations are removed once the comments have been checked. This can be used to put the comments after the declarations, like so:

```
void func();
//.< Some function

//. My enumeration
enum Foo {
  One, //.< The first enumerator
  Two, //.< The second
  Three //.< The third
};
```

Note that it does not work for function parameters. To document parameters, enable the ’javatags’ processor and use "@param" tags in your comments.

4.4.8. javatags

Extracts java-style @tags from the ends of comments, and stores them in the tags() list of the Comment object. Each tag is a CommentTag object. The tags may be inspected by the formatters or other processors.

You will need to use this processor if you want to use java-style @tags, since the HTML formatter no longer looks for them itself.

4.4.9. summary

This processor joins all the comments together for each declaration then attempts to make a summary of the comment from the first sentence. The summary is stored with the comment. It is used by the formatters extensively so you will almost always want this processor.

4.5. EmptyNS

This operation walks the AST looking for empty modules/namespaces and removes them. A namespace is empty even if it just has Forward declarations in it. Note that any types referring to these forward declarations will still have the correct name, and will probably point to the correct (non-forward) declarations anyway since Unduplicator is usually before EmptyNS.

4.6. AccessRestrictor

If the config option max_access is set, then this operation walks the AST and removes any class members without enough accessibility.

4.7. LanguageMapper

This simple operation, which is not in the list of linker_operations by default, moves all global declarations into modules based on their language. This means you can mix declarations of different languages (e.g: IDL and C++), and if one references the other (perhaps due to NameMapper) it will look like: IDL::Fresco::WidgetKit instead of just Fresco::WidgetKit. It will also prevent clashes if you have both an IDL Fresco::WidgetKit and a C++ Fresco::WidgetKit.

4.8. XRefCompiler

This operation doesn't act on the AST, but performs a peripheral operation that takes place as part of the linker process. It does use the AST, but only to find a list of source filenames.

The XRefCompiler should only be used once in a build: it reads (text formatted) xref files from the C++ parser, and writes a single compiled (Pickled) file for the formatter to use. The pickled xref file can also be used for things like CGI scripts. The xref files contain information on all* references to all symbols, including types, functions, function calls and variables.

* the completeness of the xref info is dependant on the C++ parser. At time of writing it does a pretty good job if you're not using templates, and a mediocre job when it encounters templated types or complexly overloaded functions.

The format of the output xref file, should you want to use it for a CGI script or some such, is as follows:

```
file = (data, index)
data = dict<scoped targetname, target_data>
index = dict<name, list<scoped targetname>>
target_data = (definitions : list<target_info>,
    func calls : list<target_info>,
    references : list<target_info>)
target_info = (filename, line number : int, scoped context name)
```

4.8.1. XRefCompiler options

The XRefCompiler has its own config sub-object under the Linker config object. The options are:

xref_path

A string with a "%s" in it, that will be used to find the (text) xref files for each source file in the AST. The %s is replaced with the filename of each source file in turn, and the xref info loaded from the resulting filename. The default is "./%s-xref". Note that if you specified absolute filenames, the filenames inserted will be absolute - the default will not work in that case.

xref_file

The output filename. This is a single, pickled, output file. You probably want to set the same filename in the XRefPages HTML options.

no_locals

Boolean flag. By default the xref info includes information about local variables defined inside function implementations. Setting this flag to true will cause this info to be discarded from the output file. The AST has no info about local variables in any case, so they would only appear in the XRefPages and the source file pages.

4.9. Linker options

The options for all the Linker operations except XRefCompiler are listed here:

verbose (boolean)

Verbosity flag. For config files, this attribute is set by the constructor, but only if ’verbose’ was passed as a config option.

Can be set to true from the command line with "-Wl,-v"

operations (list of strings)

If set, overrides the default list of operations to use in the Linker stage. Note that all operations except LanguageMapper and XRefCompiler do nothing without other options being set, so the default list is:

[’Unduplicator’, ’Stripper’, ’NameMapper’, ’Comments’, ’EmptyNS’, ’AccessRestrictor’]

Note that the operations are executed in order.

strip (list of strings)

If set, causes the Stripper operation to strip prefixes from names in the list, and remove declarations which don’t match any prefix. Typical usage just has one prefix. Each prefix is a scoped name, with the scopes separated by "::".

Can be set on the command line with, e.g.: "-Wl,-s,Synopsis::Parser::C++"

mapper_list (list of: strings or 2-tuples of strings)

(This feature will be moved to an actual Operation in future releases and cleaned up).

mapper_list specifies a list of either builtin mappers or external mappers that map the names of Unknown types. An Unknown type is one that refers to a type which is not a declaration in the AST. For example, if you parse your C++ CORBA object but not the skeleton/stub .hh files, then any reference to a Fresco::POA_Graphic type object will be Unknown (since there is no Fresco::POA_Graphic in the AST). The builtin mapper "C++toIDL" transforms any unknown type name starting with POA_ to remove the POA_ and change the type to IDL, and any type name ending with _ptr to remove the _ptr and change the type to IDL. See the Mixed demo for an example of this in action.

The elements of the list may be a string, in which case it must be one of the builtin mappers (there is only one: C++toIDL currently), or it may be a 2-tuple of ("module name or filename.py", "object in module"). The object in the module must have a "map" function or method that takes one argument: the Unknown type (see Synopsis.Core.Type.Unknown). The module name must be importable, which means it must be accessible from the PYTHON_PATH. A filename can be anywhere but must end in ".py".

Can be set on the command line with, e.g.: "-Wl,-m,module" where module is either a module name or a filename.

...or with, e.g.: "-Wl,-M,C++toIDL" to specify a builtin module.

max_access (integer)

If set, removes any declaration that doesn’t meet a level of visibility. This can be used to prune private or protected declarations from the AST so they wont be in the output. Values are:

1 : protected and private declarations removed

2 : private declarations removed

These actually follow the levels defined in Synopsis.Core.AST - levels higher than the given are removed (default is 0, public is 1, protected is 2, private is 3).

map_declaration_names (2-tuple or 2-list of name, type)

If set, causes all declarations to be mapped into a module of the given name and type. The name must be a scoped name, with "::" separating scopes. The type is the type of the modules used to extend the declarations into the given scope.

For example, the Synopsis RefManual uses this option to move all the C++ declarations into the Synopsis.Parser.C++ python package with this option:

map_declaration_names = ’Synopsis::Parser::C++’, ’Package’

Package is used as the type since the Python AST has Packages for Synopsis, Parser and C++. (Note they are AST.Module objects, with type() set to "Package").

All types referring to these declarations are renamed as appropriate.

comment_processors (list of strings)

If set, overrides the default setting of ['summary']. Gives a list of comment processors to apply, in order, to the AST. See the Comments section above for a list of these and what they do.

Chapter 5. The HTML Formatter

The HTML formatter is so comprehensive compared to the other formatters that it deserves a chapter of its own. See the next chapter for information about the other formatters available.

5.1. Basics

The basic operation of the HTML formatter is that it takes an AST as input, and outputs a whole bunch of .html and .png files to an output directory. These files can either be all in the one directory or ordered into nested subdirectories (see the file_layout option). The output is created by individual modules called "Pages", and like other parts of Synopsis you specify a list of these and they are executed in order. Each page generates a type of output, such as a frames index, the pages for classes and namespaces, an inheritance graph, the tree of modules to go in the corner frame, the index of individual modules to go in the left frame, etc.

The order of pages is generally not important, but there are some effects: pages compete for index pages, such as which page gets to write the "index.html" page, which page gets to be the default page for the top-left frame, which page gets to be default for the left frame, etc. The Page which asks for one of these first, gets it, which is why FramesIndex usually comes first.

5.2. Layouts

The default layout is to have three frames:

Top-left "contents" frame

> This frame shows a tree heirarchy of the modules or files in the project. Clicking on a link opens that module or file in the "index" frame below.

Bottom-left "index" frame

> This frame shows an index of the currently selected module or file, but only shows child classes, structs, namespaces or modules. When a module page is loaded in this frame and you have JavaScript enabled, a more detailed page will open in the main content area. For file pages you will need to click the "File Details" link.

Right "main" frame

> This frame shows the main documentation. At the top of each page you can see a list of documentation areas you can visit: The Modules and Files open in the top-left frame, the Inheritance Graph/Tree and Name Index load in the main frame. For "scope pages" (i.e. the pages for classes and namespaces, including the global namespace which is the default page) the page is divided into summary and detail sections: the summary shows all the declarations in this class or namespace, with a summary for each. For declarations with more comments than just the summary, the name of the declaration will be highlighted, and clicking on it will take you to its detailed information further down the page.

This three-frame layout is inspired by JavaDoc, which is similar, but turns out to be less useful than it might be since most projects don’t have a nice list of packages to go in the top-left frame like the Java libraries do. You will notice from the Synopsis RefManual however that an extensive Python project can fill the frame quite well.

To enable the three-frame layout, the FramesIndex page must be included as the first Page. You should also have at least one of ModuleListing and FileListing, and ModuleIndexer and FileIndexer to fill in the frames.

If you don’t want frames then don’t include the FramesIndex page. Whatever Page comes first will get to have the "index.html" page. For example, if ScopePages is first, then the index.html page will be the Global Namespace page. If DirBrowse is first, then it will be the listing of the project directory.

5.3. Pages

The output modules you can choose from are detailed below. Some generate multiple HTML files and others just one. Some use enough options that they have their own configuration sub-object, whereas others just use options in the main config object. Remember you can include or exclude any of these pages by setting the "pages" config option to a list of their names.

Each page module extends from the Page base class, which provides both an interface to use and some default functionality (opening closing files, setting filenames and titles, etc). You can write your own Page class if one of the provided ones doesn’t suit you, or extend from an existing one.

5.3.1. FramesIndex

This page should always go first if you want frames, else it should not be included at all. There are no options to set.

5.3.2. ScopePages

This is the most complex output Page, creating the pages for each class and namespace/module/package. It walks the whole AST creating pages as it goes using flexible "Parts" and "FormatStrategies".

For each Scope in the AST, the output page has a number of Parts. You can write your own Parts, or change the order, but the usual four are ’Heading’, ’Summary’, ’Inheritance’ and ’Detail’. A Part is responsible for visiting all the nodes in the Scope and formatting the sections for each type of declaration. It also provides methods for referencing other declarations with correct URLs, and things like formatting Types for display (with URLs for any names in the type). Each part has a list of FormatStrategies which it calls in turn for each declaration it processes, to generate the output HTML for that declaration.

The difference between the four Parts is what declarations they process (pass to the formatters) on each page:

- The Heading part only processes the scope itself, and provides things like a page title, inheritance graph, and comments for the class or module.
- The Summary part processes all the children of the scope, using a set of FormatStrategies to put them in a table with summary comments. Section headings are displayed for each type of declaration (methods, variables, etc).
- The Inheritance part displays the methods inherited from base classes, taking care to hide overridden methods.
- The Detail part processes all the children which have detailed information, using a set of FormatStrategies to list them sequentially with full comments and more detailed info (e.g.: for enums and long typedefs, exception specifications, etc).

You can control the output by changing the FormatStrategies used by each part, writing your own FormatStrategies, or even your own Parts!

The FormatStrategies included with Synopsis are:

Heading

Formats the heading of the page for the given declaration. For classes, it shows the class name, any template parameters, a link to the file the class was in, and the module/namespace name. The module/namespace name is placed in a DIV element with class "class-namespace" that by default is floated in the top-right corner of the page.

ClassHeirarchySimple

Displays the class hierarchy around this class in a textual manner. Mainly for use if you don’t have/want graphviz (dot).

ClassHeirarchyGraph

Displays the class hierarchy around this class, including all parents, using the Dot formatter to create an embedded PNG image.

SummaryAST

Formats the elements of the AST in a Summary manner. That is, two columns, with the left column for the type and the right column for the name and other details. SummaryAST actually prints the column divider (a TD element). Declarations which will appear in the Detail section are linked to automatically where the name of the declaration is (so you click on the name and it takes you to the detail). For declarations with no detailed info, the name is not hyperlinked and the #name link will be to the summary.

DetailAST

Formats the elements of the AST in a Detailed manner. The declaration is printed out similar to how it would in code, and the comments and other details (e.g.: values for an enum) are shown expanded.

SummaryCommenter

> Formats the comments of the declaration using a CommentFormatter instance, and puts them in a SPAN element of class ’summary’, which in the default Synopsis CSS is italic.

DetailCommenter

> Formats the comments of the declaration using a CommentFormatter instance, and puts them in a DIV element of class ’desc’.

Inheritance

> A simple formatter that just prints the name of the declaration with a link to its documentation, intended for use in the Inheritance Part.

SourceLinker

> Formats a link to the source code for this declaration as "[Source]". The source code is actually generated by the FileSource Page, so make sure you include that if you want to use this FormatStrategy, and vice versa.

XRefLinker

> Formats a link to the cross-reference information for this declaration as "[XRef]". The xref info is actually generated by the XRefPages Page, so make sure you include that if you want to use this FormatStrategy, and vice versa.

The default lists of strategies for each "Part" are:

Heading

> Heading, ClassHierarchyGraph, DetailCommenter

Summary

> SummaryAST, SummaryCommenter

Inheritance

> Inheritance

Detail

> DetailAST, DetailCommenter

5.3.2.1. ScopePages Options

The ScopePages Page is the most complex to configure, since its layout is so flexible. Options are set in the sub-object "ScopePages" inside the HTML config object:

```
# Normal HTML config
class HTML (Base.Formatter.HTML):
    # A sub object for ScopePages config
    class ScopePages:
        # Following options go here...
```

The following options are available:

parts

A list of Part objects to use on each page. Each list item is an import specification with the base package "Synopsis.Formatter.HTML.ASTFormatter.", giving the following default parts:

```
parts = ['Heading', 'Summary', 'Inheritance', 'Detail']
```

heading_formatters

A list of formatters to use in the Heading Part. Each list item is an import specification with the base package 'Synopsis.Formatter.HTML.FormatStrategy.', allowing you to specify any of the FormatStrategies listed earlier.

The default is:

```
heading_formatters = ['Heading', 'ClassHierarchyGraph', 'DetailCommenter']
```

summary_formatters

A list of formatters to use in the Summary Part. Each list item is an import specification with the base package 'Synopsis.Formatter.HTML.FormatStrategy.', allowing you to specify any of the FormatStrategies listed earlier.

The default is:

```
summary_formatters = ['SummaryAST', 'SummaryCommenter']
```

inheritance_formatters

A list of formatters to use in the Inheritance Part. Each list item is an import specification with the base package 'Synopsis.Formatter.HTML.FormatStrategy.', allowing you to specify any of the FormatStrategies listed earlier.

The default is:

```
inheritance_formatters = ['Inheritance']
```

detail_formatters

A list of formatters to use in the Detail Part. Each list item is an import specification with the base package 'Synopsis.Formatter.HTML.FormatStrategy.', allowing you to specify any of the FormatStrategies listed earlier.

The default is:

```
    detail_formatters = ['DetailAST', 'DetailCommenter']
```

5.3.3. ModuleListing

This page creates a single output file with a tree of all packages/modules/namespaces in the AST. Each package/module/namespace is a link to an index file of that package, module or namespace.

See the tree_formatter option for how to control the formatting of the tree.

You can change the types of AST Module node which are included in this tree by creating a "ModuleListing" subobject and setting the option "child_types". For example, the Python parser creates "package" and "module" types, IDL creates "module" types, and C++ creates "namespaces".

The ModuleListing Page creates a link at the top of all pages called "Modules" that is also shown in the top-left frame. You can change this text using the "short_title" option of the ModuleListing subobject.

5.3.3.1. Options

The following options can be set for ModuleListing:

child_types

A list of module types to show in the listing. Each type should be the string from the 'type()' method of the Module objects. Examples are 'module', 'namespace', and 'package'. By setting this you can restrict what modules are shown, such as only showing Packages in Python. The default is to print all Module nodes in the AST.

short_title

String that sets the short title of the Page that is displayed in all other Pages as a link to the ModuleListing. The default is 'Modules'.

E.g.: To only show packages and change the title:

```
# Normal HTML config
class HTML (Base.Formatter.HTML):
    # A sub object for ModuleListing config
    class ModuleListing:
        child_types = ['package'] # only show packages
        short_title = 'Packages'  # set the title to match
```

5.3.4. ModuleIndexer

This page goes hand in hand with ModuleListing, creating a concise listing of the declarations in each package/module/namespace. Each declaration is linked to its documentation, but clicking on another package/module/namespace will also load its index page in this frame if the user has javascript enabled. The declarations are sorted into types and displayed in their sections, just like in ScopePages.

5.3.5. FileListing

This page creates a single output file with a tree of all source files used to create the AST. Each file links to an index page of declarations in that file.

See the tree_formatter option for how to control the formatting of the tree.

5.3.6. FileIndexer

Goes hand in hand with FileListing, creating the concise pages for each file.

If you are using the FileSource Page, a [File Source] link will automatically be included at the top of the page.

If you are using the FileDetails Page, a [File Details] link will automatically be included at the top of the page.

Unlike ModuleIndexer, the declarations are displayed in the order they were in the file, with appropriate indenting for non-global declarations.

5.3.7. FileDetails

Shows details about each file, such as what files it included and a quick list of all the declarations declared in it.

5.3.8. FileSource

Shows source code for each file that the C++ parser created SXR info for. The actual markup of the source code is done by a C++ module for speed. The stored SXR info is used to hyperlink each identifier in the source code with a link to the documentation for that variable, class, etc, and also provide a hover text of what the identifier is, e.g: "member variable LinkStore::private::buffer_start" or "local variable main". The ability to correctly identify the identifiers is dependent on the C++ parser, which is currently pretty good except for sometimes getting overloaded functions wrong, and not understanding template instantiations too well just yet.

If the FileSource page is used should be included in the list before RawFilePages, since both compete to provide the source code for each file - and you generally want the syntax highlighted version!

5.3.8.1. Options

The options for FileSource are set in the FileSource config sub-object:

links_path

> A string containing a single "%s". When the "%s" is replaced by the filename it should give the filename of the "links" file generated by the C++ parser for that input file. The default is "./%s" but you could also use "syn/%s" or "syn/%s.syn" to give two examples.

toc_files

> A list of strings, each being a .TOC file to use for linking source code to declarations. The current AST is automatically linked (see toc_from below).

scope

> A string which is prepended to all names before being looked up in the (combined) TOC. For example, the Synopsis RefManual uses "Synopsis::Parser::C++::" since all the C++ types reside in that part of the AST. This is needed since the names used for lookup are generated by the C++ parser, but the types in the TOC (including the current AST) may have been changed by the Linker.

toc_from

> A string which names the Page to get TOC information from. The default is to use the HTML formatter option 'default_toc'. By setting this to 'ScopePages' or 'XRefPages' you can force AST declarations in the source file pages to link to either the ScopePage for that declaration or the XRef page for that declaration.

Note that in previous versions there was also a file_path option, but since version 0.5 it has been superseded by storing the original filenames in the AST along with the "base"-relative filenames.

5.3.9. RawFilePages

Shows source code for each file in the project as unformatted text. See DirBrowse for info on how "files in the project" are found.

5.3.10. DirBrowse

Shows directories in the project, where each file links to the source code from FileSource if available, the formatted page from RawFilePages if available, or is not a link if neither is available.

5.3.10.1. Options

For both DirBrowse and RawFilePages the following options can be used at the HTML configuration level (i.e.: there is no sub-object, since they are used by more than one Page).

base_dir

String that names the directory which is the "base" of the project, i.e., the one which all project files are named from. If you want to include the name of your project in the directory names, set base_dir to the directory containing your project directory. Relative paths can be used - they must be relative from the current directory when Synopsis is run.

start_dir

String that names the directory which will be the start of the recursive directory searching. This should be below the base_dir.

exclude_globs

List of strings. Each string is a "glob" expression (* for wildcard, ? for single-character wildcard). Any file or directory matching the glob expression will not be listed by DirBrowse.

5.3.11. XRefPages

Creates pages with cross-reference information for each declaration. Each declaration will show its parent scope, child declarations (if it’s a class or module), references and places it was called from. You can navigate around the XRef documentation, but there are no comments shown so it is more useful as a navigation aide for either the source code or regular ScopePages documentation.

The XRefPages module loads compiled XRef information from a single file given in the xref_file option. The XRef information is originally generated in raw (text) format by the C++ parser. You should use the XRefCompiler Linker operation to compile the xref info.

5.3.11.1. Options

The following options are available. They must be set in the XRefPages sub-object:

xref_file

The filename of the compiled xref info. This option should be the same as the xref_file option of the XRefCompiler, unless you are running Synopsis from different directories. Synopsis cannot load the text based output from the C++ parser here - you must specify a compiled xref file.

link_to_scopepages

Boolean flag, which if set causes extra links to be inserted to the ScopePages entry for each declaration in the AST. Note that some declarations might not be in the AST, f.ex. if they are local variables.

5.3.12. InheritanceGraph

Creates a single page with all inheritance graphs as embedded images using dot (graphviz) to create the graphs. Classes that have no subclasses or superclasses are not shown for brevity.

5.3.12.1. Options

The follwing options are available. They must be set in the InheritanceGraph sub-object:

direction

A string direction to use for laying out the graph nodes. The default is ’vertical’, but setting this to ’horizontal’ causes the hierarchies to be laid out left-to-right. This can be useful if you have a relatively "flat" class hierarchy that otherwise is much wider than most browser windows.

5.3.13. InheritanceTree

Creates a single page with the entire object hierarchy in a tree structure. This page is a text-only alternative to the InheritanceGraph page, but both can be used without problems.

5.3.14. NameIndex

Creates a single page with an index of all class, variable, function/method and namespace/module names. The names are sorted alphabetically.

5.4. Options

Other options than those mentioned above for specific pages are:

verbose

Boolean flag. If set to true then extra info is printed to the console at various stages. For example, timing information will be shown for each Page, allowing you to see which Pages are taking the longest.

pages

A list of import specifications for the Pages to use. The default attribute is "htmlPageClass" and the default package is "Synopsis.Formatter.HTML.". The default value of "pages" is: [’ScopePages’, ’ModuleListing’, ’ModuleIndexer’, ’FileListing’, ’FileIndexer’, ’FileDetails’, ’InheritanceTree’, ’InheritanceGraph’, ’NameIndex’, ’FramesIndex’] sorter

An import specification for a Sorter object to use for sorting declarations in many places (eg: ScopePages). The default is to use the builtin class Synopsis.Formatter.HTML.ScopeSorter, else you can write your own with the same interface.

datadir

String that sets the data directory containing Synopsis’ images. This is usually set as a default option in Config.py by configure, so that a Synopsis installed in /usr will default to /usr/share/synopsis.

stylesheet

String that sets the destination filename name of the stylesheet file used in all generated pages. Synopsis output relies heavily on CSS for formatting. The default is "style.css".

stylesheet_file

String that sets the source filename of the file to copy the stylesheet from. The default is ’$prefix/share/synopsis/html.css’.

comment_formatters

This option sets a list of comment formatters to use for formatting AST comments. The comments should have already been processed in the Linker stage to extract @tags and create summarys - the formatters are responsible for converting the processed info into HTML output. The formatters are applied in turn to the comment text, in the order they are listed. Available formatters are:

none

Dummy formatter, does nothing.

ssd

Dummy formatter, does nothing. The old "ssd" formatter is now replaced by the "ssd" comment processor in the Linker stage.

java

Dummy formatter, does nothing. The old "java" formatter is now replaced by the "java" comment processor in the Linker stage.

quotehtml

Quotes any HTML that may be in the comments. Use this if your comments include un-quoted characters like angle-brackets. Using this formatter will prevent you from using any HTML tags in your comments.

summary

Dummy formatter, does nothing. The old "summary" formatter is now replaced by the "summary" comment processor in the Linker stage.

javadoc

Formats any attributes that were processed in the Linker stage in a JavaDoc style. Currently recognised tags are: @return, @param, @see, and @attr.

qtdoc

Attempts to recognise QtDoc style tags (\sa, \param, \return). Currently broken.

section

Looks for any empty lines in the comments and replaces them with paragraph breaks. In effect, replaces any two consecutive newlines with a close paragraph followed by an open paragraph. The whole comment is also prefixed with an open paragraph and suffixed with a close paragraph.

The default is [’javadoc’, ’section’]

toc_out

Specifies a filename to write a TOC (Table Of Contents) file to. The TOC is an index listing the (relative) URL for each declaration in the AST. You can use the TOC to, for example, link usage of a library to the documentation of that library.

toc_in

List of strings, specifies a list of filenames to read TOC files from. You may optionally specify a prefix for each TOC file, by adding a ’|’ (pipe) character followed by the prefix to the end of the filename. For example, if you are specifying a TOC file for the "Foo" project, which has documentation available online at "http://foo.org/doc/", you can specify:

```
toc_in = [ ’Foo.toc|http://foo.org/doc/’ ]
```

This will cause all URLs in the Foo.toc file to be prefixed with the given prefix.

default_toc

String which specifies the default page to generate TOC info from. The default is ’ScopePages’. The page specified by this option will be asked to generate the TOC for the whole AST. The TOC is used internally to generate the default links for all AST declarations - on source pages, scope pages, XRef pages, module and file listings, etc. You can set this to ’XRefPages’ to default to showing XRef info for the declaration. No other Page can be used. See the ’toc_from’ option of the FileSource Page for a similar option.

tree_formatter

Import specification which specifies the object to use for formatting trees in the ModuleListing and FileListing pages. The base package for the import is ’Synopsis.Formatter.HTML.’. The default is ’TreeFormatter.TreeFormatter’.

file_layout

Import specification which specifies the FileLayout object to use. The default is ’Synopsis.Formatter.HTML.FileLayout’, but you can also specify ’Synopsis.Formatter.HTML.NestedFileLayout’ to put the output in different directories depending on the Page and Namespace/Module.

output_dir

String which specifies the directory to output the HTML files into. The directory does not need to exist beforehand.

structs_as_classes

A boolean flag, which if true causes C++ structs to be listed in the same sections as the classes. Use this if your project uses structs similarly to classes and you want them to be displayed together.

page_format

An import specification which specifies the format to use for generating all output pages. The default package is ’Synopsis.Formatter.HTML.Page.’. The default is ’PageFormat’, but you may also specify ’TemplatePageFormat’ to use a template file for each output file.

To use the TemplatePageFormat, create a template HTML file with the formatting you require. Ensure that the string "@TITLE@" is in the <title> element, and the string "@CONTENT@" is where you want to the Synopsis output to go. Then add a TemplatePageFormat config sub-object to the HTML config and set these options:

file

Specifies the template filename to use

copy_files

Specifies a list of files that will be copied to the output directory. Use this for things like images. Each entry in the list is a 2-tuple giving the original filename and final filename.

An example taken from the C++ demo is:

```
class HTML (Config.Base.HTML):
    class TemplatePageFormat:
        file = ’template-sxr.html’
        copy_files = [(’gapbuffer-logo.png’, ’logo.png’)]
```

In addition, you may use the following command line arguments. Except for "-o", they must all be set using the "-Wf,foo" syntax.

-o filename

Specifies an output directory. You do not need to use special syntax for this option.

-v

Enables verbose operation.

-d datadir

Sets the data directory where images and stylesheets will be loaded from by default.

-s stylesheet

Sets the name of the stylesheet in the generated manual directory (not the original file!)

-S stylesheet_file

Sets the filename of the stylesheet to copy into the generated manual directory.

-n namespace

Sets the namespace ???

-c comment_formatter

Specifies an additional comment formatter to use. The given comment_formatter must be either an object name accessible from the available PYTHON_PATH, or a filename with global functions matching the requirements.

-C comment_formatter

Specifies an additional inbuilt comment formatter to use. See the CommentFormatter section above for a list of available comment formatters.

-t toc_filename

Specifies that the TOC should be written to the given filename.

-r toc_filename

Specifies that the TOC should be read from the given filename. The read TOC is merged with the current TOC.

Chapter 6. Other Formatters

Synopsis has other formatters than the HTML formatter, although they are far less developed.

6.1. HTML_Simple

This formatter simply writes one output file containing the whole AST, marked up in HTML.

The HTML_Simple formatter does not support any config options yet, however some command line options are still supported. Don't forget to set them using the "-Wf,-foo" syntax!

-o filename

: Specifies an output filename. The default is to send output to the console (stdout). You do not need to use the special syntax for this option.

-v

: Enables verbose operation.

-s stylesheet

: Specifies the stylesheet to link to in the generated HTML file. If note specified, then no stylesheet is linked to.

6.2. DocBook

This formatter attempts to format the output as DocBook format. Unfortunately the elements available in DocBook do not cover all the C++ types.

The DocBook formatter does not support any config options yet, however some command line options are still supported. Don't forget to set them using the "-Wf,-foo" syntax!

-o filename

: Specifies an output filename. The default is to send output to the console (stdout). You do not need to use the special syntax for this option.

-v

: Enables verbose operation.

-d

: Specifies that no DOCTYPE element should be output.

-m

Specifies that the (Synopsis Reference -) Manual Format should be used. This format doesn’t try to use the docbook code elements, and instead uses nested sections and titles to identify elements.

6.3. BoostBook

This formatter attempts to format the output as BoostBook format. BoostBook is a derivative of DocBook with full support of C++ constructs, developed by the Boost project for documenting their libraries.

The DocBook formatter does not support any config options yet, however some command line options are still supported. Don’t forget to set them using the "-Wf,-foo" syntax!

-o filename

Specifies an output filename. The default is to send output to the console (stdout). You do not need to use the special syntax for this option.

-v

Enables verbose operation.

6.4. TexInfo

This formatter attempts to output TexInfo .info files. These can be compiled using other tools into man-pages, PDF files, info manuals, etc.

The DocBook formatter does not support any config options yet, however some command line options are still supported. Don’t forget to set them using the "-Wf,-foo" syntax!

-o filename

Specifies an output filename. The default is to send output to the console (stdout). You do not need to use the special syntax for this option.

-v

Enables verbose operation.

6.5. Dot

This formatter uses the ’dot’ tool, a part of GraphViz, to generate graphs of the AST. It is mostly used by the HTML formatter for generating inheritance graphs, but can also be used as a stand-alone formatter. It can currently generate two types of diagram: inheritance and single. The inheritance diagram shows the entire hierarchy, whereas the single diagram shows only the hierarchy starting at a given class, and going one level down and all the way up to the top-level parent class(es). The former is used by the InheritanceGraph HTML page, and the latter by the individual class pages.

The Dot formatter does not support any config options yet, however some command line options are still supported. Don’t forget to set them using the "-Wf,-foo" syntax!

-o filename

Specifies an output filename. You do not need to use the special syntax for this option.

-v

Enables verbose operation.

-t title

Specifies a title for the graph

-i

Specifies an "inheritance" style graph, showing all classes.

-s

Specifies a "single" style graph, showing only the graph related to a single class.

-O

Causes operations to be shown

-A

Causes attributes to be shown

-f format

Specifies an output format, can be one of:

dot

GraphViz dot file.

ps

PostScript file.

png

Portable Network Graphic file, for use in web pages.

gif

Graphics Interchange Format, for use in web pages (prefer png due to patent issues).

map

HTML map segment, for embedding in web pages with links to classes.

html

HTML segment, for embedding in web pages with links to classes or as a standalone page.

-r tocfile

Specifies a TOC file to read, allowing the generation of links to the classes in the graph. See the HTML formatter for more info about TOC files.

-R classname

Specifies the origin of the graph for Single graphs

-d direction

Specifies the direction. Can be ’horizontal’ or ’vertical’.

6.6. ASCII

This formatter attempts to output the AST in a format resembling C++.

The ASCII formatter does not support any config options yet, however some command line options are still supported. Don’t forget to set them using the "-Wf,-foo" syntax!

-o filename

Specifies an output filename. The default is to send output to the console (stdout). You do not need to use the special syntax for this option.

-v

Enables verbose operation.

-b

Shows comments in bold using terminal codes

-c number

Shows comments in the given colour number, using terminal codes.

6.7. DUMP

This formatter outputs the AST in a format useful for debugging Synopsis or your config. It displayes the AST hierarchy using a generic object dumper. It also shows the types dictionary and the listing of sourcefiles. The best way to use the DUMP formatter is to pipe the output into "less", preferably with the "-r" flag so that the bold names are shown. Using the "-r" flag however will cause problems with comments longer than a screen width.

The DUMP formatter does not support any config options yet, however some command line options are still supported. Don’t forget to set them using the "-Wf,-foo" syntax! Note that if none of -d, -t or -f are specified, the default is equivalent to -d and -t.

-o filename

Specifies an output filename. The default is to send output to the console (stdout). You do not need to use the special syntax for this option.

-v

Enables verbose operation.

-d

Shows the declarations, following the AST hierarchy

-t

Shows the types dictionary

-f

Shows forward declarations

6.8. Dia

This formatter attempts to output the AST as a Dia graph. The Dia file format is just XML, so this makes the formatter quite simple.

The Dia formatter does not support any config options yet, however some command line options are still supported. Don’t forget to set them using the "-Wf,-foo" syntax!

-o filename

Specifies an output filename. It should probably end in ".dia". You do not need to use the special syntax for this option.

-v

Enables verbose operation.

-m

Hides the operations/methods in the classes.

-a

Hides the attributes/variables in the classes.

-p

Hides the parameters in the operation/method specifications.

Chapter 7. Customising Synopsis

Synopsis is by design highly modular. At the top level, you can add new parsers and formatters. At the middle level you can add or modify existing linker operations or HTML pages. At the low level you can set a multitude of options for almost all modules. This chapter lists some of the more common modifications.

7.1. Importing Objects and Import Specifications

Several config options in this manual use import specifications for importing custom objects at various places. Synopsis has a utility function called import_object which can import an object from a file or from an import path.

To do this you write a ’spec’, instead of just a module name. The import_object function decides what to do with the spec, depending on extra parameters given to it (base package and default attribute). Wherever you see mention of an import specification in this manual, it will also say whether a base package or default attribute is given to the import_object function.

’spec’ must be either a string or a tuple of two strings:

if spec is a tuple (or list) of two strings

> A tuple of two strings means load the module from the first string, and look for an attribute using the second string. The default attribute and base package specifications make no difference in this case.
>
> E.g.: spec is (’config.py’, ’MyModule’), import_module loads the ’config.py’ file, then looks for the MyModule object inside that and returns it.

if spec is a string

> One string is interpreted according to the optional arguments.
>
> if default attribute given
>
> > Load the module and look for the default attribute in the module and return that.
> >
> > E.g.: default attribute is ’htmlPageClass’, spec is ’MyPage.py’, import_object loads MyPage.py and then looks for an htmlPageClass attribute in the file.
>
> if base package given
>
> > Prepend the base package and then import the module and return that.

E.g.: base package is ’Synopsis.Formatter.HTML.’, spec is ’XRefPages.XRefPages’, import_object loads ’Synopsis.Formatter.HTML.XRefPages’ and returns the class XRefPages inside the XRefPages module as the loaded object.

if both base package and default attribute given

Prepend the base package and import the module, then look for the default attribute in the module and return that. Note that the last element in the spec doesn’t have to be a module... it could be a class or any object, as long as it has the default attribute in it.

E.g.: base package is ’Synopsis.Formatter.HTML.’, default attribute is ’htmlPageClass’, spec is ’XRefPages’. import_object loads ’Synopsis.Formatter.HTML.XRefPages’, looks for the htmlPageClass attribute in the loaded module, and returns that. In this example the htmlPageClass attribute is actually an alias to the XRefPages class inside the module, allowing us to easily get the class object without giving all the classes the same name, i.e.:

```
class XRefPages:
    # .. stuff ..

htmlPageClass = XRefPages
```

7.2. New output format

Making a new output formatter for Synopsis is easy. There are two structural approaches: create a new formatter and put it in the Formatter directory (for now, you cannot specify a path with the formatter), or create a stand-alone formatter that reads an AST file. The former is more integrated, but the latter doesn’t require you to put the file in the Synopsis/Formatter directory.

Most formatters follow the pattern of taking the AST, and visiting its nodes in turn while writing output. This is done via the Visitor pattern, which basically means that you create an object with one method for each type of AST node, such as classes, modules, operations (methods), typedefs, etc. You may find it easiest to take an existing formatter with similar output structure to what you want, and modify the output to fit your particular format.

One point to note is the way that Type objects are formatted. Type objects (defined in Synopsis/Core/Type.py) are used to store things like function parameters, return types, variable types, typedef types, and class parents. (A Type is used for class parents since the base class might be a parameterized instance of a class template). Types have their own class and node hierarchy, though it is simpler than the AST hierarchy. Consider the type "const char*" - this is represented as a "Modifier" Type containing the "const" and "*", which points to a "Base" Type for the "char". The Base type could instead be a Declared type, if the complete type was "const string*", for example.

The best way to figure out how a program is represented as an AST is to use the DUMP formatter. I’ve had to use it so many times myself that I lost count years ago! The DUMP formatter will show both the AST heirarchy, and also how any types you are using are represented.

The existing formatters generally format types by implementing a Visitor for the Type hierarchy. Each Type object has an "accept(self, visitor)" method which simply calls the appropriate method of the visitor object. For example, the Modifier Type’s accept method calls "visitor.visitModifier(self)".

The formatters have a series of "visit" methods, one for each Type class, each of which helps build a label string. In the above Modifier example, the visitModifier method recursively formats the aliased type (e.g: a Base Type if it was a char or a Declared Type if it was a string), then sticks a "const" on the front and a "*" on the end of the label string.

The Synopsis Reference Manual contains graphs and class details for the AST and Type hierarchies.

7.3. New parser (new language)

Making a new parser is not so simple. A parser is a complex beast! However, you might be able to find an existing parser, and retrofit it to produce a Synopsis AST. It is unlikely (but possible!) though that you will be able to fit all the language features into the Synopsis AST hierarchy. In that case, modifying the AST is definately an option, though not a simple one.

7.4. New linker operation

Making a new linker option is much easier. You can even use the import facility to put your new operation in any file you want (e.g.: your config.py file).

The operation interface is very simple; it has one method: execute(self, ast) which returns nothing (the AST is modified in-place). Within your execute method, you can do whatever you like to the AST. For example you can rename methods or classes, simplify function parameter types, check the comments and perform actions based on those, etc.

7.5. New HTML Page

You can add a new type of HTML output to an existing HTML setup by creating or modifying a Page object. Like Linker Operations, the HTML Pages are specified with import specifications, allowing you to easily write your own. See the Page class for a list of the interface methods you can implement, and the base methods you can use. The mundane stuff like opening files and creating headers is all done for you, so you only have to take the AST and make the interesting bits!

A lot of the customizability of the HTML formatter is managed automatically, by having the Pages register themselves for various things:

Page.register(self)

> Override this method to do the registration

config.set_contents_page(self, page)

> Call this with the name of your main page (string ending in ".html") to register for the contents page, which is the top-left frame. If you are the first page to call this method, you will get to be the contents page - and keep it.

config.set_index_page(self, page)

> Call this with the name of your main page (string ending in ".html") to register for the index page, which is the left frame. If you are the first page to call this method, you will get to be the index page - and keep it.

config.set_main_page(self, page)

> Call this with the name of your main page (string ending in ".html") to register for the main page, which is the main frame, or start page if you're not using frames. If you are the first page to call this method, you will get to be the main page - and keep it. Note that if you get to be the main page, the FileLayout will return "index.html" for your main page (the one you gave to this method) for most methods of FileLayout.

PageManager.addRootPage(self, file, label, target, visibility)

> Call this to add your page to the list of main pages at the top of every generated page. file is the filename to be linked to, label is the text to show, target is the frame to display in ("contents", "index", "main", or "_top" for a new window). visibility is 1 for just the main pages, 2 for all pages (including the smaller ones like ModuleListing).

Page.register_filenames(self, start)

> Override this method to register filenames based on the target AST. "start" is the start node of the AST; typically the global namespace, but may be some other scope based on configuration and command line options.

PageManager.register_filename(self, filename, page, scope)

> Call this for each file you can generate to register yourself for generating that file. This function is used by the FileSource and RawFilePages pages to compete for who gets to generate the output for that file. Use the filename_info() method to check if you got it or not. "scope" is any data you need to find out what makes that filename. It is called "scope" for the general case of it being an AST node (of which only Scopes usually have pages!). This facility is also used by the GUI to generate files on the fly when the user clicks a link! You should only call this method from the register_filenames() method of your Page class, not from the register() method!

The best way to figure this out is to study the existing Pages.