Synopsis Tutorial

Stefan Seefeld

Synopsis Tutorial

Stefan Seefeld

Version 0.11

Table of Contents

List of Examples

Chapter 1. Introduction

Synopsis is a source code introspection tool. It provides parsers for a variety of programming languages (C, C++, Python, IDL), and generates internal representations of varying granularity. The only *stable* representation, which is currently used among others to generate documentation, is an Abstract Semantic Graph.

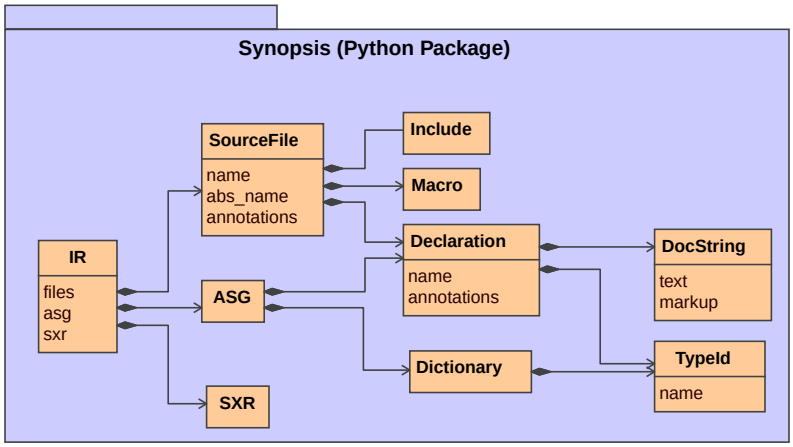
This tutorial is focussed on the ASG and the concepts around it. Other representations are presently being worked on, notably in relation to the C++ parser. To learn more about those (Parse Tree, Symbol Table, etc.) see the Developer's Guide[1].

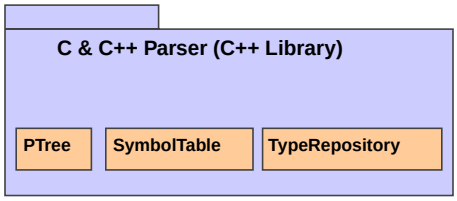
1.1. Inspecting Code

1.2. Internal Representations

Synopsis parses source code into a variety of *internal representations* (IRs), which then are manipulated in various ways, before some output (such as a cross-referenced API documentation) is generated by an appropriate *formatter*.

At the core of Synopsis are a set of programming-language independent IRs which all *parser frontends* generate. One of these representations is the *Abstract Semantic Graph*, which stores declarations and their relationships. Another is the *SXR* Symbol Table, which stores information about symbols and their use in the source code. Other representations exist (such as the C++ Parse Tree), but they are not yet stored in a publicly accessible form.





For details about the ASG, see Section 3.1, “The ASG”

[1] ../DevGuide/index.html

At this time, the C++ frontend's IRs (PTree, SymbolTable, etc.) are not yet accessible through python, though they eventually will be, making it possible to use Synopsis as a source-to-source compiler. To learn more about the evolving C & C++ parser and its IRs, see the Developer's Guide[2].

1.3. Documenting Source-Code

Being read and understood is at least as important for source code as it is for it to be processed by a computer. Humans have to maintain the code, i.e. fix bugs, add features, etc.

Therefor, typically, code is annotated in some form in that adds explanation if it isn't self-explanatory. While comments are often used to simply disable the execution of a particular chunk of code, some comments are specifically addressed at readers to explain what the surrounding code does. While some languages (e.g. Python) have built-in support for *doc-strings*, in other languages ordinary comments are used.

Typically, comments are marked up in a specific way to discriminate documentation from ordinary comments. Further the content of such comments may contain markup for a particular formatting (say, embedded HTML).

Example 1.1. Typical C++ code documentation

C++ may contain a mix of comments, some representing documentation.

```
//! A friendly function.
void greet()
{
  // FIXME: Use gettext for i18n
  std::cout << "hello world !" << std::endl;
}
       \
```

In Synopsis all declarations may be annotated. C and C++ parsers, for example, will store comments preceding a given declaration in that declaration's `annotations` dictionary under the key `comments`. Later these comments may be translated into documentation (stored under the key `doc`), which may be formatted once the final document is generated.

Translating comments into doc-strings involves the removal of comment markers (such as the `//!` above), as well as the handling of processing instructions that may be embedded in comments, too.

For languages such as Python such a translation isn't necessary, as the language has built-in support for documentation, and thus the parser itself can generate the 'doc' annotations.

[2] ../DevGuide/index.html

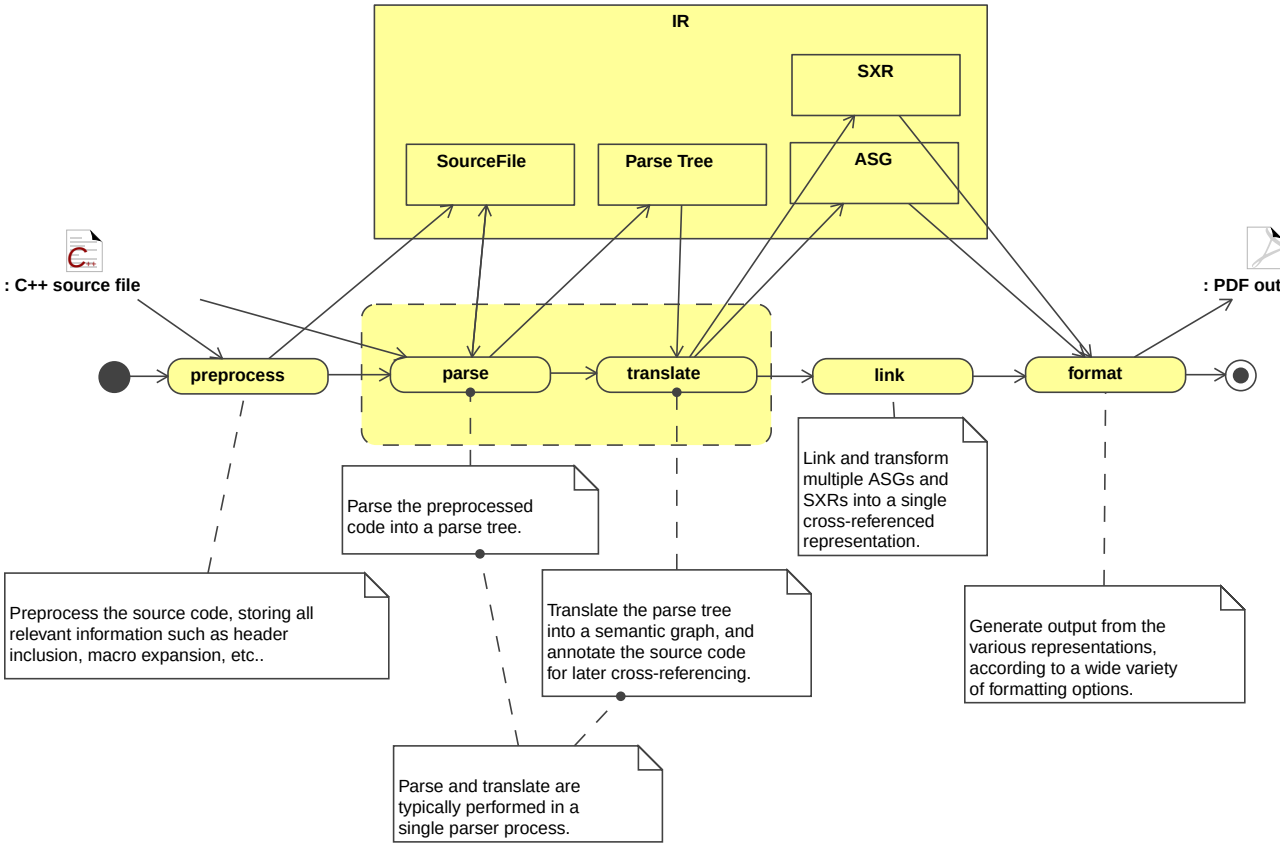
Example 1.2. Python code documentation

Python has support for documentation built into the language.

```
>>> def greet():
...     """The greet function prints out a famous message."""
...     print 'hello world !'
...
>>>help(greet)


Help on function greet in module __main__:

greet()
    The greet function prints out a famous message.
        \
```

1.4. The Synopsis Processing Pipeline

Synopsis provides a large number of *processor* types that all generate or operate on data extracted from source code. Parsers parse source code from a variety of languages, linkers combine multiple IRs, resolving cross-references between symbols, and formatters format the ASG into a variety of output media.



A typical processing-pipeline to generate API Documentation with source-code cross-references.

All these Processor types share a common design, to make it easy to combine them into pipelines, and add custom processors. For more documentation about this architecture, see Section 3.3, “Composing A Pipeline”.

Chapter 2. Using the synopsis tool

In this section we are going to explore the possibilities to generate documentation from source code. We will demonstrate how to use synopsis standalone as well as in conjunction with existing build systems. Further, we will see how to adapt synopsis to your coding and commenting style, as well as how to generate the output in a format and style that fulfills your needs.

2.1. Option Handling

The synopsis tool combines three optional types of processors: parsers (specified with the `-p` option), linker processors (specified with the `-l` option, and formatters (specified with the `-f` option). If a parser is selected, any input is interpreted as source files of the respective language. Otherwise it will be read in as a stored IR. Similarly, if a formatter is selected, output is generated according to the formatter. Otherwise it will contain a stored IR.

For all of the three main processors, arguments can be passed down using the `-W`. For example, to find out what parameters are available with the Cxx parser, use the `--help` option:

```
$ synopsis -p Cxx -h
Parameters for processor 'Synopsis.Parsers.Cxx.Parser':
   profile               output profile data
   cppflags              list of preprocessor flags such as -I or -D
   preprocess            whether or not to preprocess the input
   ...
      \
```

Then, to pass a `preprocess` option, either of:

```
synopsis -p Cxx -Wp,--preprocess ...
```

```
synopsis -p Cxx -Wp,preprocess=True ...
```

The first form expects an optional string argument, while the second form expects a python expression, thus allowing to pass python objects such as lists. (But be careful to properly escape characters to get the expression through the shell !)

But passing options via the command line has its limits, both, in terms of usability, as well as for the robustness of the interface (all data have to be passed as strings !). Therefor, for any tasks demanding more flexibility a scripting interface is provided, which will be discussed in the next chapter.

2.2. Parsing Source-code

Let's assume a simple header file, containing some declarations:

```
#ifndef Path_h_
#define Path_h_

//. A Vertex is a 2D point.
struct Vertex
{
  Vertex(double xx, double yy): x(xx), y(yy) {}
```

```
  double x; //.< the x coordinate
  double y; //.< the y coordinate
};

//. Path is the basic abstraction
//. used for drawing (curved) paths.
class Path
{
public:
  virtual ~Path() {}
  //. Draw this path.
  virtual void draw() = 0;
  // temporarily commented out...
  // bool intersects(const Path &);
private:
};

#endif

        \
```

Process this with

```
synopsis -p Cxx -f HTML -o Paths Path.h
```

to generate an html document in the directory specified using the `-o` option, i.e. `Paths`.

The above represents the simplest way to use **synopsis**. A simple command is used to parse a source-file and to generate a document from it. The parser to be used is selected using the `-p` option, and the formatter with the `-f` option.

If no formatter is specified, synopsis dumps its internal representation to the specified output file. Similarly, if no parser is specified, the input is interpreted as an IR dump. Thus, the processing can be split into multiple synopsis invocations.

Each processor (including parsers and formatters) provides a number of parameters that can be set from the command line. For example the Cxx parser has a parameter `base_path` to specify a prefix to be stripped off of file names as they are stored in synopsis' internal representation. Parser-specific options can be given that are passed through to the parser processor. To pass such an option, use the `-Wp,` prefix. For example, to set the parser's `base_path` option, use

```
synopsis -p Cxx -Wp,--base-path=<prefix> -f HTML -o Paths Path.h
```

2.3. Emulating A Compiler

Whenever the code to be parsed includes *system headers*, the parser needs to know about their location(s), and likely also about *system macro* definitions that may be in effect. For example, parsing:

```
#include <vector>
#include <string>

typedef std::vector<std::string> option_list;
      \
```

requires the parser to know where to find the `vector` and `string` headers.

Synopsis will attempt to emulate a compiler for the current programming language. By default, **`synopsis -p Cxx`** will try to locate c++ or similar, to query system flags. However, the compiler can be specified via the `--emulate-compiler` option, e.g. **`synopsis -p Cxx -Wp,--emulate-compiler=/usr/local/gcc4/bin/g++`**.

All languages that use the Cpp processor to preprocess the input accept the `emulate-compiler` argument, and pass it down to the Cpp parser. See Section 4.3, “The Cpp Parser” for a detailed discussion of this process.

2.4. Using Comments For Documentation

Until now the generated document didn't contain any of the text from comments in the source code. To do that the comments have to be translated first. This translation consists of a filter that picks up a particular kind of comment, for example only lines starting with "//.", or javadoc-style comments such as "/**...*/", as well as some translator that converts the comments into actual documentation, possibly using some inline markup, such as Javadoc or ReST.

The following source code snippet contains java-style comments, with javadoc-style markup. Further, an embedded processing instruction wants some declarations to be grouped.

```
#ifndef Bezier_h_
#define Bezier_h_

#include "Path.h"
#include <vector>

namespace Paths
{

/**
 * The Bezier class. It implements a Bezier curve
 * for the given order.
 */
template <size_t Order>
class Bezier : public Path
{
public:
  /** Create a new Bezier.*/
  Bezier();

  /** @group Manipulators {*/

  /**
   * Add a new control point.
   * @param p A point
   */
  void add_control_point(const Vertex &);

  /**
   * Remove the control point at index i.
   * @param i An index
```

```
   */
  void remove_control_point(size_t i);
  /** }*/
  virtual void draw();
private:
  /** The data...*/
  std::vector<Vertex> controls_;
};

}

#endif

        \
```

The right combination of comment processing options for this code would be:

```
synopsis -p Cxx --cfilter=java --translate=javadoc -lComments.Grouper ...
```

The `--cfilter` option allows to specify a filter to select document comments, and the `--translate` option sets the kind of markup to expect. The `-l` option is somewhat more generic. It is a *linker* to which (almost) arbitrary post-processors can be attached. Here we pass the Comments.Grouper processor that injects Group nodes into the IR that cause the grouped declarations to be documented together.

Chapter 3. Scripting And Extending Synopsis

Often it isn't enough to provide textual options to the synopsis tool. The processors that are at the core of the synopsis framework are highly configurable. They can be passed simple string / integer / boolean type parameters, but some of them are also composed of objects that could be passed along as parameters.

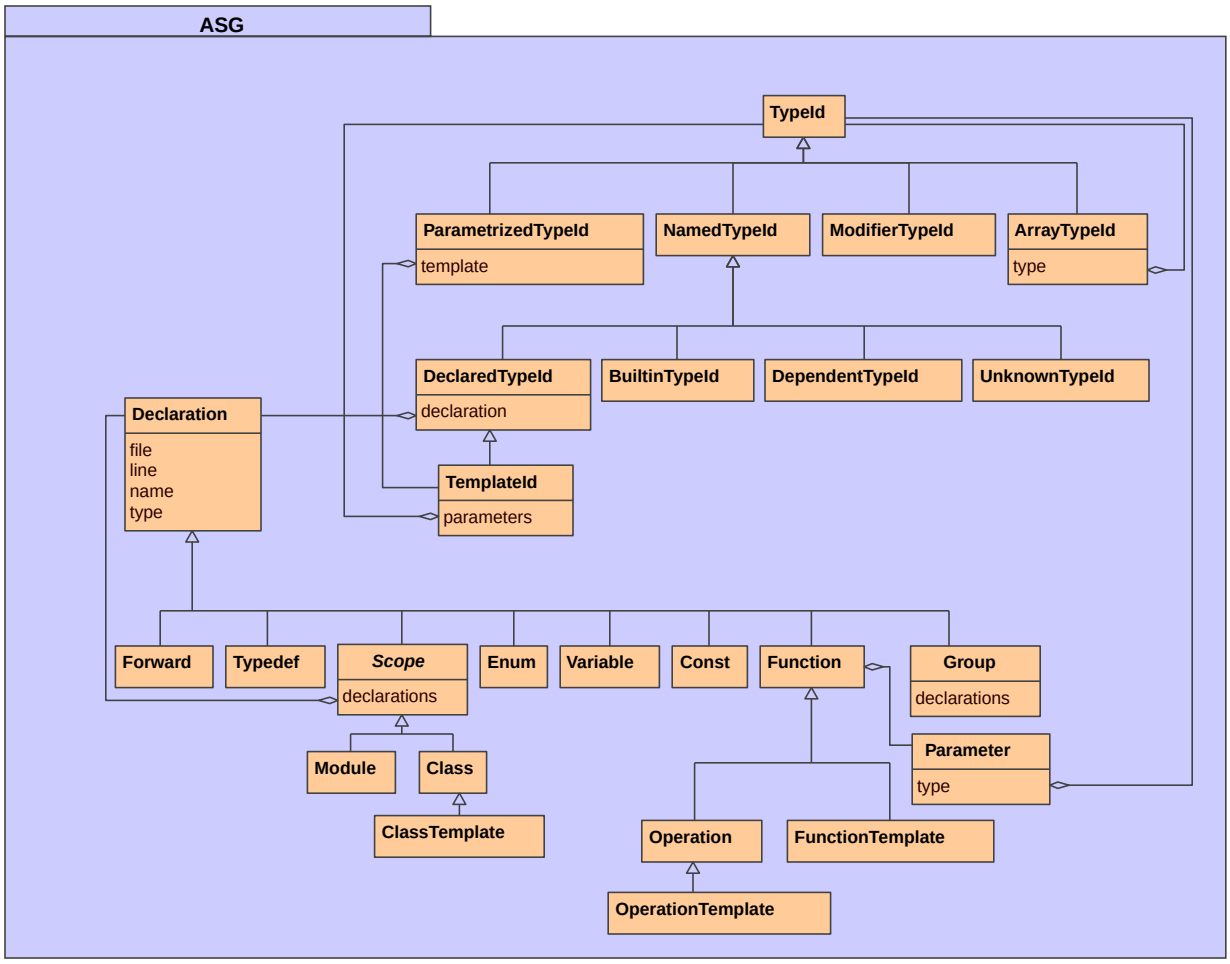
While synopsis provides a lot of such building blocks already, you may want to extend them by subclassing your own.

In all these cases scripting is a much more powerful way to let synopsis do what you want. This chapter explains the basic design of the framework, and demonstrates how to write scripts using the built-in building blocks as well as user extensions

3.1. The ASG

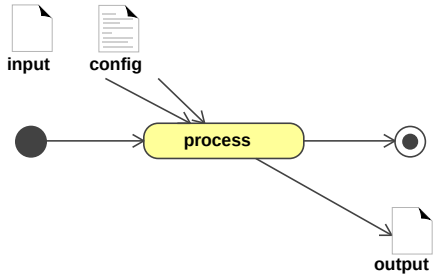
At the core of synopsis is a representation of the source code to be analyzed called an Abstract Semantic Graph (ASG). Language-specific syntax gets translated into an abstract graph of statements, annotated with all the necessary metadata to recover the important details during further processing.

At this time only one particular type of statements is translated into an ASG: declarations. This can be declarations of types, functions, variables, etc. Attached to a declaration is a set of comments that was found in the source code before the declaration. It is thus possible to provide other metadata (such as code documentation) as part of these comments. A variety of comment processors exist to extract such metadata from comments.



3.2. The Processor class

The Processor class is at the core of the Synopsis framework. It is the basic building block out of which processing pipelines can be composed.



The requirement that processors can be composed into a pipeline has some important consequences for its design. The process method takes an `ir` argument, which it will operate on, and then return. It is this `ir` that forms the backbone of the pipeline, as it is passed along from one processor to the next. Additionally, parameters may be passed to the processor, such as input and output.

```
def process(self, ir, **keywords):

  self.set_parameters(keywords)
  self.ir = self.merge_input(ir)

  # do the work here...

  return self.output_and_return_ir()
```

Depending on the nature of the processor, it may parse the input file as source code, or simply read it in from a persistent state. In any case, the result of the input reading is merged in with the existing asg.

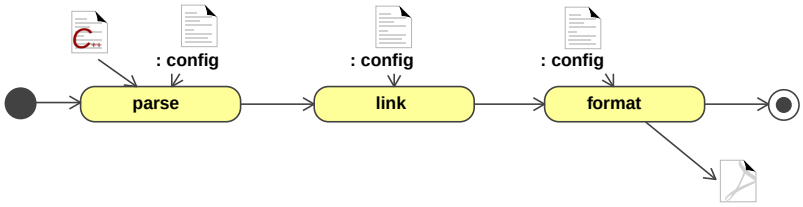
```
def process(self, ir, **keywords):

  self.set_parameters(keywords)

  for file in self.input:
    self.ir = parse(ir, file))

  return self.output_and_return_ir()
```

Similarly with the output: if an output parameter is defined, the ir may be stored in that file before it is returned. Or, if the processor is a formatter, the output parameter may indicate the file / directory name to store the formatted output in.

```
def process(self, ir, **keywords):

  self.set_parameters(keywords)
  self.ir = self.merge_input(ir)

  self.format(self.output)

  return self.ir
```

3.3. Composing A Pipeline

With such a design, processors can simply be chained together:

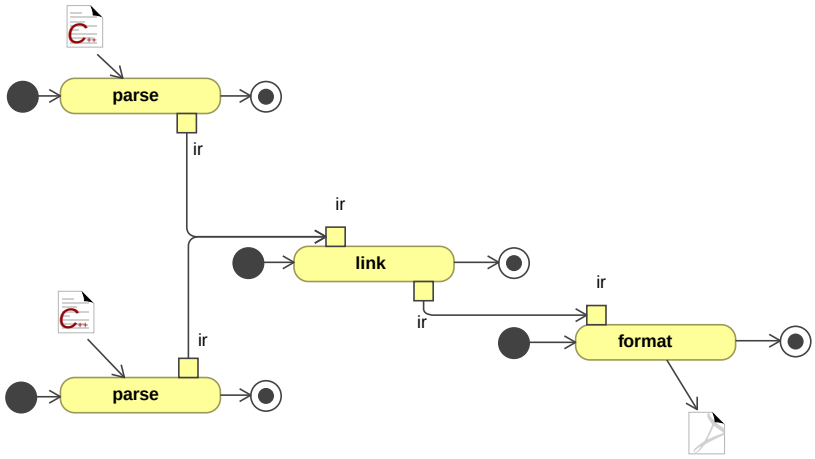


A parser creates an IR, which is passed to the linker (creating a table of contents on the fly) which passes it further down to a formatter.

```
parser = ...
linker = ...
formatter = ...
```

```
ir = IR()
ir = parser.process(ir, input=['source.hh'])
ir = linker.process(ir)
ir = formatter.process(ir, output='html')
```

And, to be a little bit more scalable, and to allow the use of dependency tracking build tools such as make, the intermediate IRs can be persisted into files. Thus, the above pipeline is broken up into multiple pipelines, where the 'output' parameter of the parser is used to point to IR stores, and the 'input' parameter of the linker/formatter pipeline then contains a list of these IR store files.



Parse `source1.hh` and write the IR to `source1.syn`:

```
parser.process(IR(), input = ['source1.hh'], output = 'source1.syn')
```

Parse `source2.hh` and write the IR to `source2.syn`:

```
parser.process(IR(), input = ['source2.hh'], output = 'source2.syn')
```

Read in `source1.syn` and `source2.syn`, then link and format into the `html` directory:

```
formatter.process(linker.process(IR(), input = ['source1.syn', \
'source2.syn']), output = 'html')
```

3.4. Writing your own synopsis script

The synopsis framework provides a function `process` that lets you declare and expose processors as commands so they can be used per command line:

```
#
# Copyright (C) 2006 Stefan Seefeld
# All rights reserved.
# Licensed to the public under the terms of the GNU LGPL (>= 2),
# see the file COPYING for details.
#

from Synopsis.process import process
from Synopsis.Processor import Processor, Parameter, Composite
```

```
from Synopsis.Parsers import Cxx
from Synopsis.Parsers import Python
from Synopsis.Processors import Linker
from Synopsis.Processors import Comments
from Synopsis.Formatters import HTML
from Synopsis.Formatters import Dot
from Synopsis.Formatters import Dump

class Joker(Processor):

    parameter = Parameter(':-)', 'a friendly parameter')

    def process(self, ir, **keywords):
        # override default parameter values
        self.set_parameters(keywords)
        # merge in IR from 'input' parameter if given
        self.ir = self.merge_input(ir)

        print 'this processor is harmless...', self.parameter

        # write to output (if given) and return IR
        return self.output_and_return_ir()

cxx = Cxx.Parser(base_path='../src')

ss = Comments.Translator(filter = Comments.SSFilter(),
                         processor = Comments.Grouper())
ssd_prev = Comments.Translator(filter = Comments.SSDFilter(),
                           processor = Composite(Comments.Previous(),
                                                 Comments.Grouper()))
javadoc = Comments.Translator(markup='javadoc',
                              filter = Comments.JavaFilter(),
                              processor = Comments.Grouper())
rst = Comments.Translator(markup='rst',
                          filter = Comments.SSDFilter(),
                          processor = Comments.Grouper())

process(cxx_ss = Composite(cxx, ss),
        cxx_ssd_prev = Composite(cxx, ssd_prev),
        cxx_javadoc = Composite(cxx, javadoc),
        cxx_rst = Composite(cxx, rst),
        link = Linker(),
        html = HTML.Formatter(),
        dot = Dot.Formatter(),
        joker = Joker(parameter = '(-;'))

        \
```

With such a script `synopsis.py` it is possible to call

```
python synopsis.py cxx_ssd --output=Bezier.syn Bezier.h
        \
```

to do the same as in Chapter 2, *Using the synopsis tool*, but with much more flexibility. Let's have a closer look at how this script works:

3.4.1. Importing all desired processors

As every conventional python script, the first thing to do is to pull in all the definitions that are used later on, in our case the definition of the `process` function, together with a number of predefined processors.

3.4.2. Composing new processors

As outlined in Section 3.3, “Composing A Pipeline”, processors can be composed into pipelines, which are themselfs new (composite) processors. Synopsis provides a Composite type for convenient pipeline construction. Its constructor takes a list of processors that the process method will iterate over.

3.4.3. Defining New Processors

New processors can be defined by deriving from Processor or any of its subclasses. As outlined in Section 3.2, “The Processor class”, it has only to respect the semantics of the `process` method.

3.4.4. Exposing The Commands

With all these new processrs defined, they need to be made accessible to be called per command line. That is done with the `process` function. It sets up a dictionary of named processors, with which the script can be invoked as

```
python synopsis.py joker
          \
```

which will invoke the joker's `process` method with any argument that was provided passed as a named value (keyword).

Chapter 4. Processor Design

4.1. The Python Parser

The Python parser expects Python source files as input, and compiles them into an Abstract Semantic Graph. Note that directory names are valid input, too, if they correspond to Python packages, i.e. have an `__init__.py` file in them. At this time, this compilation is based purely on static analysis (parsing), and no runtime-inspection of the code is involved.

This obviously is obviously only of limitted use if objects change at runtime.

The found docstrings are identified and attached to their corresponding objects. If a `docformat` specifier is provided (either in terms of a `__docformat__` variable embedded into the Python source or the definition of the parser's `default_docformat` parameter, this format is used to parse and format the given docstrings.

Here are the available Python-Parser parameters:

primary_file_only	If false, in addition to the primary python file imported modules are parsed, too, if they are found.
base_path	A prefix (directory) to strip off of the Python code filename.
sxr_prefix	If this variable is defined, it points to a directory within which the parser will store cross-referenced source code. This information may be used to render the source code with cross-references during formatting.
default_docformat	Specify the doc-string format for the given python file. By default doc-strings are interpreted as plaintext, though other popular markup formats exist, such as ReStructuredText (`rst`), or JavaDoc (`javadoc`)

4.2. The IDL Parser

The IDL parser parses CORBA IDL.

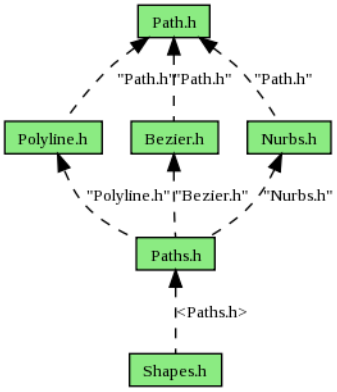
4.3. The Cpp Parser

The Cpp parser preprocesses IDL, C, and C++ files. As any normal preprocessor, it will generate a file suitable as input for a C or C++ parser, i.e. it processes include and macro statements. However, it will store the encountered preprocessor directives in the ASG for further analysis.

As the list of included files may grow rather large, two mechanisms exist to restrict the number of files for which information is retained. The primary_file_only parameter is used to indicate that only the top-level file being parsed should be included. The base_path parameter, on the other hand, will restrict the number files if main_file_only is set to False. In this case, the base_path is used as a prefix, and only those file whose name starts with that prefix are marked as main.

For each included file, a SourceFile object is created and added to the parent's Include list. Further, all macro declarations, as well as macro calls, are recorded. While most useful in conjunction with the C and

Cxx processors, these data can be of use stand-alone, too. For example consider a tool that reports file dependencies based on #include statements. The Dot formatter (see Section 4.10, “The Dot Formatter”) can generate a file dependency graph from the Cpp processor output alone:



Whenever the code to be parsed includes *system headers*, the parser needs to know about their location(s), and likely also about *system macro* definitions that may be in effect.

The Cpp parser provides two parameters to specify this emulation process, `emulate_compiler` and `compiler_flags`. To illustrate their use, let us probe for the system flags that get generated:

```
> synopsis --probe -p Cpp -Wp,--emulate-compiler=g++
Compiler: g++
Flags:
Language: C++
Header search path:
  /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../include/c++/4.1.2
  ...
  /usr/include

Macro definitions:
  __STDC__=1
  __cplusplus=1
  ...
    \
```

Sometimes it isn't enough to have the compiler name itself. Some flags may modify the header search path, or the predefined macros. For example, GCC can be instructed not to consider system headers at all:

```
> synopsis --probe -p Cpp -Wp,--emulate-compiler=g++ \
-Wp,compiler-flags=[\"-nostdinc\"]
Compiler: g++
Flags: -nostdinc
Language: C++
Header search path:


Macro definitions:
  __STDC__=1
  __cplusplus=1
```

```
...
  \
```

Here, the set of predefined header search paths is empty. Note, that the `--compiler-flags` option (which, as you may remember, maps to the `compiler_flags` processor parameter) expects a (Python) list. Therefor, we use the form without the leading dashes, so we can pass Python code as argument (See Section 2.1, “Option Handling” for details), with appropriate quoting.

For details about the parameters see Section B.3, “Synopsis.Parsers.Cpp.Parser”.

4.4. The C Parser

The C parser parses C.

The C parser parses C source-code. If the preprocess parameter is set, it will call the preprocessor (see Section 4.3, “The Cpp Parser”). It generates an ASG containing all declarations.

4.5. The Cxx Parser

The Cxx parser parses C++. If the preprocess parameter is set, it will call the preprocessor (see Section 4.3, “The Cpp Parser”). Its main purpose is to generate an ASG containing all declarations. However, it can store more detailed information about the source code to be used in conjunction with the HTML parser to generate a cross-referenced view of the code. The sxr_prefix parameter is used to indicate the directory within which to store information about the source files being parsed.

4.6. The Linker

The Linker recursively traverses the ASG using the Visitor pattern, and replaces any duplicate types with their originals, and removes duplicate declarations. References to the removed declarations are replaced with a reference to the original.

There are many additional transformations that may be applied during linking, such as the extraction of documentation strings from comments, the filtering and renaming of symbols, regrouping of declarations based on special annotations, etc., etc..

4.7. Comment Processors

Comments are used mainly to annotate source code. These annotations may consist of documentaton, or may contain processing instructions, to be parsed by tools such as Synopsis.

Processing comments thus involves filtering out the relevant comments, parsing their content and translating it into proper documentation strings, or otherwise perform required actions (such as ASG transformations).

Here are some examples, illustrating a possible comment-processing pipeline.

4.7.1. Comment Filters

To distinguish comments containing documentation, it is advisable to use some convention such as using a particular prefix:

```
//. Normalize a string.
std::string normalize(std::string const &);
// float const pi;
//. Compute an area.
float area(float radius);
       \
```

Using the `ssd`(read: Slash-Slash-Dot) prefix filter instructs Synopsis only to preserve those comments that are prefixed with `//.`

```
synopsis -p Cxx --cfilter=ssd ...
```

Synopsis provides a number of built-in comment filters for frequent / popular prefixes. Here are some examples:

Comment prefix	**Filter class**	**option name**
//	SSFilter	ss
///	SSSFilter	sss
//.	SSDFilter	ssd
/*...*/	CFilter	c
/*!...*/	QtFilter	qt
/**...*/	JavaFilter	java

4.7.2. Comment Translators

Once all irrelevant comments have been stripped off, the remainder needs to be transformed into proper documentation. As the actual formatting can only be performed during formatting (at which time the output medium and format is known), there are still things that can be done at this time: Since in general it isn't possible to auto-detect what kind of markup is used, a translator assists in mapping stripped comment strings to doc-strings, to which a `markup` specifier is attached. While this specifier is arbitrary, the only two values supported by the HTML and DocBook formatters are `javadoc` and `rst`(for ReStructuredText).

Note that this comment translation is specific to some programming languages (such as C, C++, and IDL). Notably Python does provide a built-in facility to associate doc-strings to declarations. (In addition, the doc-string markup can be expressed via special-purpose variable `__docformat__` embedded into Python source code.

4.7.3. Transformers

In addition to the manipulation of the comments themselves, there are actions that may be performed as a result of *processing-instructions* embedded into comments.

For example, A Grouper transformer groups declarations together, based on special syntax:

```
/** @group Manipulators {*/

/**
 * Add a new control point.
 * @param p A point
 */
void add_control_point(const Vertex &);
```

```
/**
 * Remove the control point at index i.
 * @param i An index
 */
void remove_control_point(size_t i);
/** }*/
virtual void draw();
      \
```

To process the above `@group` processing-instruction, run **`synopsis -p Cxx --cfilter=java -l Grouper ...`**

4.8. The Dump Formatter

The Dump formatter's main goal is to provide a format that is as close to the ASG tree, is easily browsable to the naked eye, and provides the means to do validation or other analysis.

It generates an xml tree that can be browsed via mozilla (it uses a stylesheet for convenient display), or it can be analyzed with some special tools using xpath expressions.

It is used right now for all unit tests.

4.9. The DocBook Formatter

The DocBook formatter allows to generate a DocBook section from the given ASG.

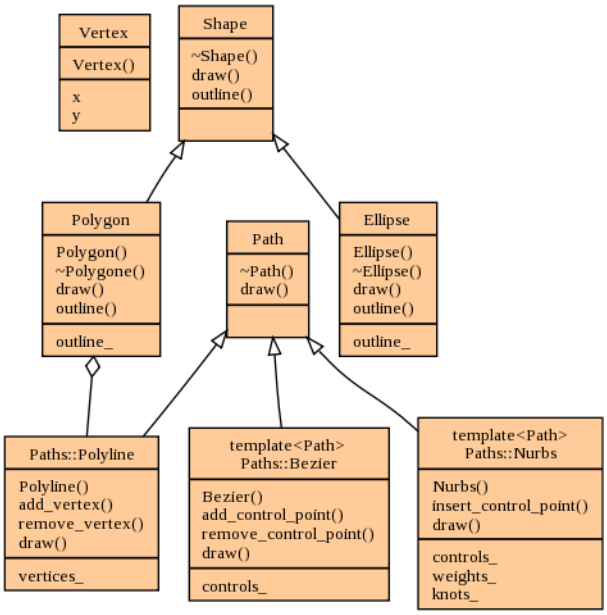
Here are the most important parameters:

title	The title to be used for the toplevel section.
nested_modules	True if nested modules are to be formatted to nested sections. If False, modules are flattened and formatted in sibling sections.
generate_summary	If True, generate a *summary* section for each scope, followed by a *details* section.
with_inheritance_graph	If True, generate `SVG` and `PNG` inheritance graphs for all classes. (use the `graph_color` option to set the background color of the graph nodes)
secondary_index_terms	If True, add `secondary` entries in `indexterms`, with the fully qualified names. This is useful for disambiguation when the same unqualified-id is used in multiple scopes.

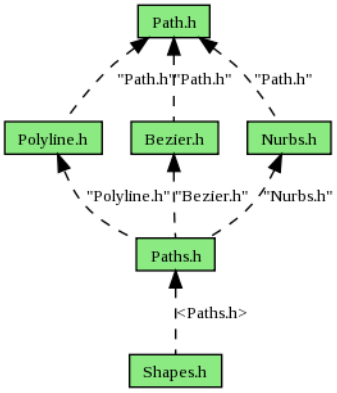
4.10. The Dot Formatter

The Dot formatter can generate graphs for various types and output formats. Among the supported output formats are png, svg, and html.

A typical use is the generation of UML class (inheritance and aggregation) diagrams:



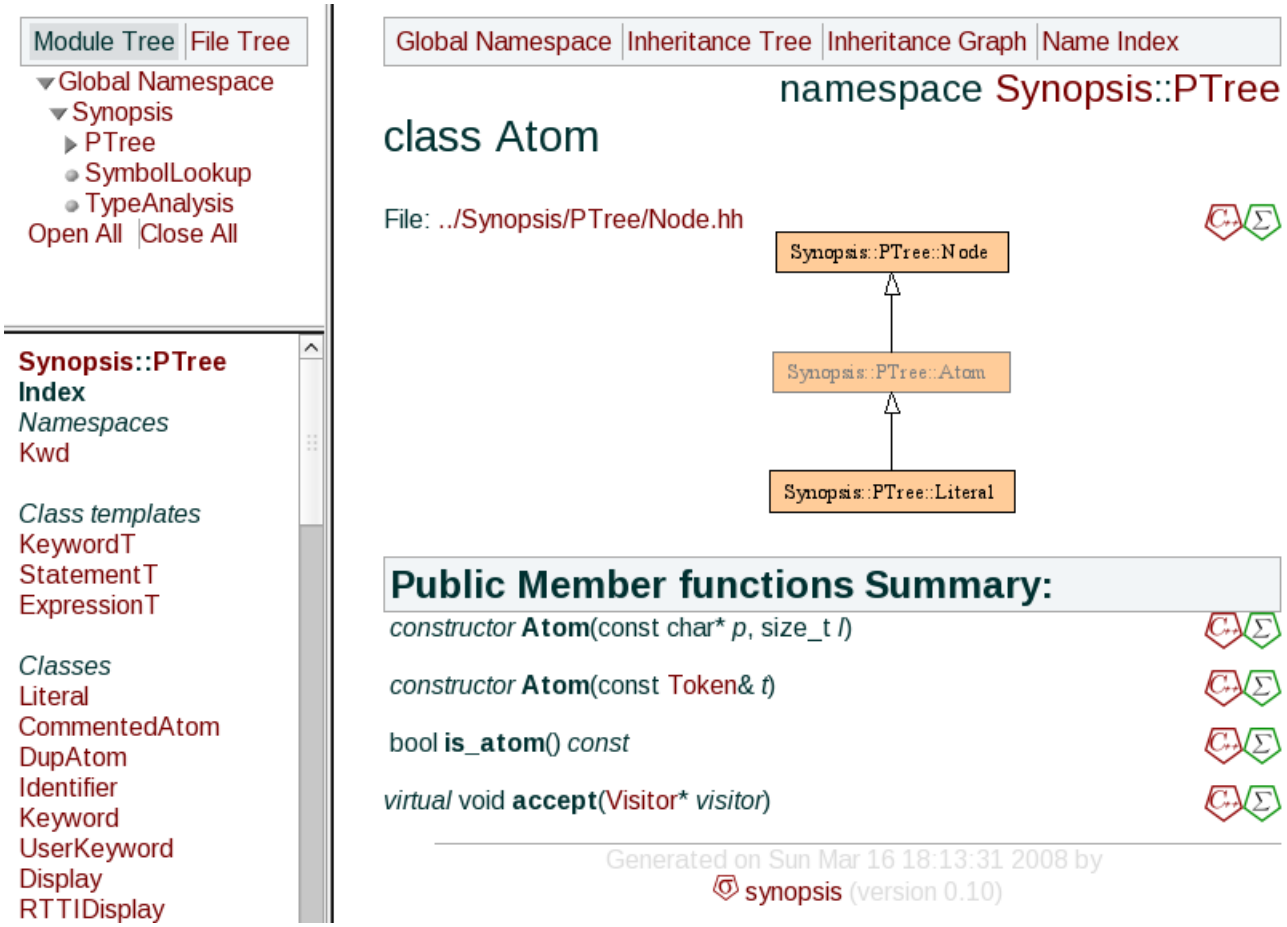
But it can also be used to generate a graphical representation of file inclusions:



4.11. The HTML Formatter

The HTML formatter generates html output. It is designed in a modular way, to let users customize in much detail how to format the data. All output is organized by a set of *views*, which highlight different aspects of data. Some views show the file / directory layout, others group declarations by scopes, or provide an annotated (and cross-referenced) source view.

By default the formatter generates its output using frames. The views are formatter parameters. `index` is a list of views that fill the upper-left index frame. `detail` is a list of views for the lower-left detail frame, and `content` sets all the views for the main content frame.



When the `index` and `detail` arguments are empty lists, non-framed html will be generated.

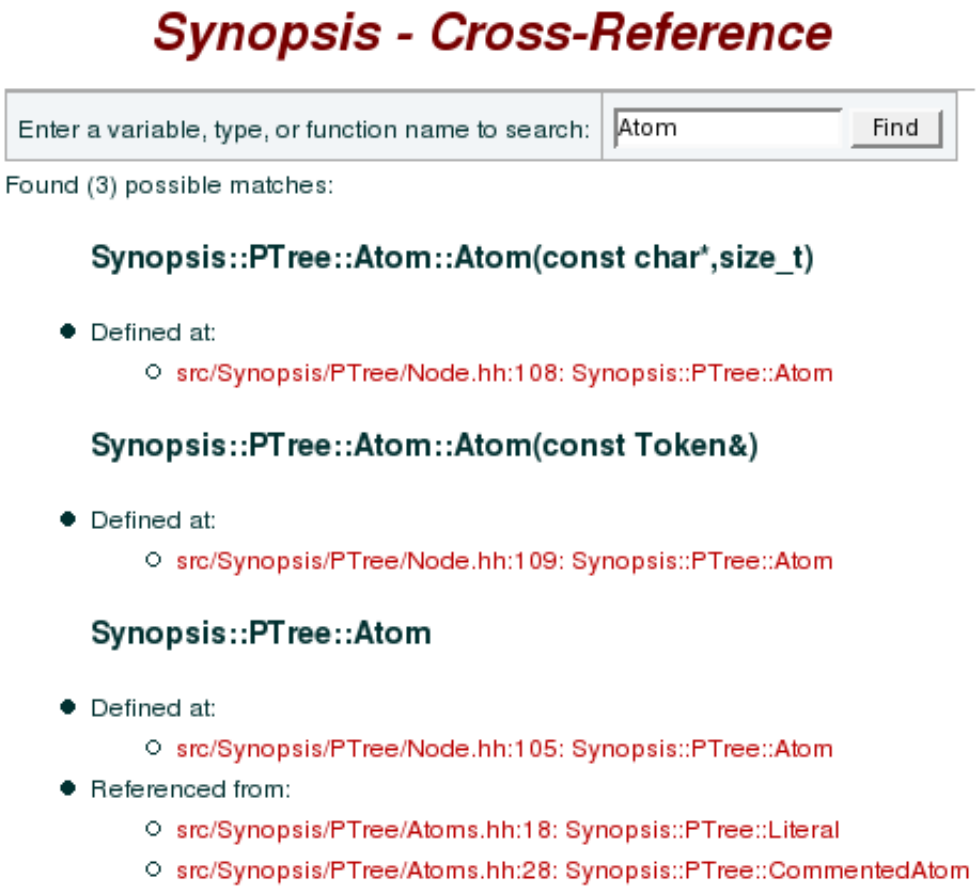
Here are the most important View types:

Scope	The most important view for documentation purposes is doubtless the Scope view. It presents all declaration in a given scope, together with a number of references to other views if appropriate.
InheritanceGraph	A UML-like inheritance diagram for all classes.
NameIndex	A global index of all declared names (macros, variables, types, ...)
Source	A cross-referenced view of a source file.
XRef	A listing of symbols with links to their documentation, definition, and reference.
FileDetails	Shows details about a given file, such as what other files are included, what declarations it contains, etc.
Directory	Presents a directory (of source files). This is typically used in conjunction with the Source view above.
FileTree	A javascript-based file tree view suitable for the index frame for navigation.

ModuleTree	A javascript-based module tree view suitable for the index frame for navigation.

4.12. The SXR Formatter

The SXR formatter is a variant of the HTML formatter. However, as its focus is not so much documentation as code navigation, there are a number of important differences. Its default set of views is different, and instead of displaying listings of all identifiers on static html, it loads a database of (typed) identifiers and provides an interface to query them.



It is to be used with an http server, either a default http server such as apache in conjunction with the *sxi.cgi* script that is part of Synopsis, or by using the *sxr-server* program. The latter performs better, as the database is kept in-process, while in case of sxi.cgi it needs to be reloaded on each query.

Appendix A. Description of program options for the synopsis executable

Appendix A. Description of program options for the synopsis executable

The synopsis executable is a little convenience frontend to the larger Synopsis framework consisting of IR-related types as well as *Processor* classes.

While the full power of synopsis is available through scripting (see Chapter 3, *Scripting And Extending Synopsis*), it is possible to quickly generate simple documentation by means of an easy-to-use executable, that is nothing more but a little script with some extra command line argument parsing.

This tool has three processor types it can call:

Parser — A processor that will parse source code into an internal abstract semantic graph (ASG). Various Parsers have a variety of parameters to control how exactly they do that.

Linker — A processor that will remove duplicate symbols, forward declarations, and apply any number of ASG manipulations you want. The user typically specifies what sub-processors to load to run from the linker.

Formatter — A processor that generates some form of formatted output from an existing ASG, typically html, docbook xml, or class graphs. Other formatters exist to assist debugging, such as a List formatter that prints specific aspects of the IR to stdout, or a Dump formatter that writes the IR to an xml file, useful for unit testing.

You can run synopsis with a single processor, for example to parse a C++ file `source.hh` and store the ASG into a file `source.syn`, or you can combine it directly with linker and or formatter to generate the output you want in a single call.

While the document generation in a single call is convenient, for larger projects it is much more sensible to integrate the document generation into existing build systems and let the build system itself manage the dependencies between the intermediate files and the source files.

For example, a typical Makefile fragment that contains the rules to generate documentation out of multiple source files may look like this:

```
    hdr := $(wildcard *.h)
    syn := $(patsubst %.h, %.syn, $(hdr))

    html: $(syn)
        synopsis -f HTML -o $@ $<

    %.syn: %.h
        synopsis -p Cxx -I../include -o $@ $<
\
```

Here is a listing of the most important available options:

-h, --help	print out help message
-V, --version	print out version info and exit
-v, --verbose	operate verbosely
-d, --debug	operate in debug mode
-o, --output	output file / directory
-p, --parser	select a parser
-l, --link	link
-f, --formatter	select a formatter
-I	set an include search path
-D	specify a macro for the parser
-W	pass down additional arguments to a processor. For example '-Wp,-I.' sends the '-I.' option to the parser.
--cfilter	Specify a comment filter (See Section 4.7.1, “Comment Filters”).
--translate	Translate comments to doc-strings, using the given markup specifier (See Section 4.7.2, “Comment Translators”).
--sxr-prefix	Specify the directory under which to store SXR info for the parsed source files. This causes parsers to generate SXR info, the linker to generate an sxr Symbol Table, and for the HTML formatter this causes Source and XRef views to be generated.
--probe	This is useful in conjunction with the `-p Cpp` option to probe for system header search paths and system macro definitions. (See Section 2.3, “Emulating A Compiler”).

Appendix B. Listing of some Processors and their parameters

This is a listing of all processors with their respective parameters that can be set as described in Section 3.4, “Writing your own synopsis script”.

B.1. Synopsis.Parsers.Python.Parser

Name	Default value	Description
profile	False	output profile data
verbose	False	operate verbosely
primary_file_only	True	should only primary file be processed
sxr_prefix	None	Path prefix (directory) to contain sxr info.
base_path	None	Path prefix to strip off of input file names.
default_docformat		default documentation format
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to

B.2. Synopsis.Parsers.IDL.Parser

Name	Default value	Description
profile	False	output profile data
cppflags	[]	list of preprocessor flags such as -I or -D
preprocess	True	whether or not to preprocess the input
verbose	False	operate verbosely
base_path		path prefix to strip off of the file names
primary_file_only	True	should only primary file be processed
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to

B.3. Synopsis.Parsers.Cpp.Parser

Name	Default value	Description
profile	False	output profile data
verbose	False	operate verbosely
language	C++	source code programming language of the given input file
compiler_flags	[]	list of flags for the emulated compiler
base_path	None	path prefix to strip off of the filenames
primary_file_only	True	should only primary file be processed
cpp_output	None	filename for preprocessed file
flags	[]	list of preprocessor flags such as -I or -D
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to
emulate_compiler		a compiler to emulate

B.4. Synopsis.Parsers.C.Parser

Name	Default value	Description
profile	False	output profile data
cppflags	[]	list of preprocessor flags such as -I or -D
preprocess	True	whether or not to preprocess the input
verbose	False	operate verbosely
sxr_prefix	None	path prefix (directory) to contain syntax info
compiler_flags	[]	list of flags for the emulated compiler
base_path		path prefix to strip off of the file names
primary_file_only	True	should only primary file be processed
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to
emulate_compiler	cc	a compiler to emulate

B.5. Synopsis.Parsers.Cxx.Parser

Name	Default value	Description
profile	False	output profile data
cppflags	[]	list of preprocessor flags such as -I or -D
preprocess	True	whether or not to preprocess the input
verbose	False	operate verbosely
sxr_prefix	None	path prefix (directory) to contain sxr info
compiler_flags	[]	list of flags for the emulated compiler
base_path		path prefix to strip off of the file names
primary_file_only	True	should only primary file be processed
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to
emulate_compiler		a compiler to emulate

B.6. Synopsis.Processors.Linker

Name	Default value	Description
profile	False	output profile data
remove_empty_modules	True	Remove empty modules.
verbose	False	operate verbosely
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to
sxr_prefix		Compile sxr data, if defined.
sort_modules	True	Sort module content alphabetically.
processors	[]	the list of processors this is composed of

B.7. Synopsis.Processors.MacroFilter

Name	Default value	Description
profile	False	output profile data
pattern		Regular expression to match macro names with.
verbose	False	operate verbosely
input	[]	input files to process
debug	False	generate debug traces

Name	Default value	Description
output		output file to save the ir to

B.8. Synopsis.Processors.Comments.Filter

Name	Default value	Description
profile	False	output profile data
output		output file to save the ir to
verbose	False	operate verbosely
debug	False	generate debug traces
input	[]	input files to process

B.9. Synopsis.Processors.Comments.Translator

Name	Default value	Description
profile	False	output profile data
concatenate	False	Whether or not to concatenate adjacent comments.
primary_only	True	Whether or not to preserve secondary comments.
verbose	False	operate verbosely
markup		The markup type for this declaration.
filter	Synopsis.Processors.Comments.Filter.SSFilter	A comment filter to apply.
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to
processor	None	A comment processor to run.

B.10. Synopsis.Formatters.Dot.Formatter

Name	Default value	Description
profile	False	output profile data
verbose	False	operate verbosely
toc_in	[]	list of table of content files to use for symbol lookup
format	ps	Generate output in format "dot", "ps", "png", "svg", "gif", "map", "html"
show_aggregation	False	show aggregation
prefix	None	Prefix to strip from all class names
input	[]	input files to process
hide_operations	True	hide operations
layout	vertical	Direction of graph
title	Inheritance Graph	the title of the graph
hide_attributes	True	hide attributes
base_url	None	base url to use for generated links
bgcolor	None	background color for nodes
debug	False	generate debug traces
output		output file to save the ir to
type	class	type of graph (one of "file", "class", "single"

B.11. Synopsis.Formatters.Dump.Formatter

Name	Default value	Description
profile	False	output profile data
show_declarations	True	output declarations
verbose	False	operate verbosely
show_types	True	output types
stylesheet	/home/stefan/projects/S10/Synopsis/../share/synopsis/dump.css	stylesheet to be referenced for rendering
show_ids	True	output object ids as attributes
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to
show_files	True	output files

B.12. Synopsis.Formatters.DocBook.Formatter

Name	Default value	Description
profile	False	output profile data
secondary_index_terms	True	add fully-qualified names to index
inline_inherited_members	False	show inherited members
verbose	False	operate verbosely
title	None	title to be used in top-level section
nested_modules	False	Map the module tree to a tree of docbook sections.
hide_undocumented	False	hide declarations without a doc-string
generate_summary	False	generate scope summaries
with_inheritance_graphs	True	whether inheritance graphs should be generated
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to
markup_formatters	{rst: Synopsis.Formatters.DocBook.Markup.RST, reStructuredText: Synopsis.Formatters.DocBook.Markup.RST, javadoc: Synopsis.Formatters.DocBook.Markup.Javadoc}	Markup-specific formatters.
graph_color	#ffcc99	base color for inheritance graphs

B.13. Synopsis.Formatters.Texinfo.Formatter

Name	Default value	Description
profile	False	output profile data
output		output file to save the ir to
verbose	False	operate verbosely
debug	False	generate debug traces
input	[]	input files to process

B.14. Synopsis.Formatters.HTML.Formatter

Name	Default value	Description
profile	False	output profile data
verbose	False	operate verbosely

Name	Default value	Description
toc_in	[]	list of table of content files to use for symbol lookup
sxr_prefix	None	path prefix (directory) under which to find sxr info
graph_color	#ffcc99	base color for inheritance graphs
input	[]	input files to process
directory_layout	Synopsis.Formatters.HTML.Direct-oryLayout.NestedDirectoryLayout	how to lay out the output files
index	[Synopsis.Formatters.HTML.Views.ModuleTree, Synopsis.Formatters.HTML.Views.FileTree]	set of index views
markup_formatters	{rst: Synopsis.Formatters.HTML.Markup.RST, reStructuredText: Synopsis.Formatters.HTML.Markup.RST, javadoc: Synopsis.Formatters.HTML.Markup.Javadoc}	Markup-specific formatters.
title	Synopsis - Generated Documentation	title to put into html header
detail	[Synopsis.Formatters.HTML.Views.ModuleIndex, Synopsis.Formatters.HTML.Views.FileIndex]	set of detail views
toc_out		name of file into which to store the TOC
stylesheet	/home/stefan/projects/S10/Synopsis/../share/synopsis/html.css	stylesheet to be used
debug	False	generate debug traces
output		output file to save the ir to
content	[Synopsis.Formatters.HTML.Views.Scope, Synopsis.Formatters.HTML.Views.Source, Synopsis.Formatters.HTML.Views.XRef, Synopsis.Formatters.HTML.Views.FileDetails, Synopsis.Formatters.HTML.Views.InheritanceTree, Synopsis.Formatters.HTML.Views.InheritanceGraph, Synopsis.Formatters.HTML.Views.NameIndex]	set of content views

B.15. Synopsis.Formatters.SXR.Formatter

Name	Default value	Description
profile	False	output profile data
src_dir		starting point for directory listing
verbose	False	operate verbosely
sxr_template	/home/stefan/projects/S10/Synopsis/../share/synopsis/sxr-template.html	html template to be used by the sxr.cgi script
title	Synopsis - Cross-Reference	title to put into html header

Listing of some Processors and their parameters

Name	Default value	Description
url	/sxr.cgi	the base url to use for the sxr cgi
sxr_prefix	None	path prefix (directory) to contain sxr info
exclude	[]	TODO: define an exclusion mechanism (glob based ?)
stylesheet	/home/stefan/projects/S10/Synopsis/../share/synopsis/html.css	stylesheet to be used
input	[]	input files to process
debug	False	generate debug traces
output		output file to save the ir to

Appendix C. Supported Documentation Markup

Synopsis can handle a variety of documentation markup through markup-formatter plugins. The most frequently used markup types are built into the framework, and are available via the **synopsis** applet. These are `Javadoc` (available as `--translate=javadoc`), and `ReStructuredText` (available as either `--translate=rst` or `--translate=reStructuredText`).

C.1. Javadoc

Synopsis provides support for Javadoc-style markup (See http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/javadoc.html). However, as Javadoc is very HTML-centric, best results will only be achieved when HTML is the only output-medium.

Javadoc comments consist of a main description, followed by tag blocks. Tag blocks are of the form `@tag`. The following block tags are recognized:
author, date, deprecated, exception, invariant, keyword, param, postcondition, precondition, return, see, throws, version

All blocks may contain any of the following inline tags, which are of the form `{@inlinetag}`:
link, code, literal

Link targets may be text, or HTML anchor elements. In case of text Synopsis interprets the it as a name-id and attempts to look it up in its symbol table.

All of the above tags are recognized and translated properly for both, the HTML as well as the DocBook formatters. Javadoc recommends to use `HTML` markup for additional document annotation. This is only supported with the HTML formatter, however.

Example C.1. C++ code snippet using Javadoc-style comments.

```
/**
 * The Bezier class. It implements a Bezier curve
 * for the given order. See {@link Nurbs} for an alternative
 * curved path class. Example usage of the Bezier class:
 * <pre>
 *   Bezier&lt;2&gt; bezier;
 *   bezier.add_control_point(Vertex(0., 0.));
 *   bezier.add_control_point(Vertex(0., 1.));
 *   ...
 * </pre>
 *
 * @param Order The order of the Bezier class.
 * @see <a href="http://en.wikipedia.org/wiki/Bezier"/>
 */
template <size_t Order>
class Bezier : public Path
{
...
     \
```

C.2. ReStructured Text

Synopsis supports the full set of ReStructuredText markup (See http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html). In order to process ReST docstrings, docutils 0.4 or higher must be installed. If Docutils is not installed, ReST docstrings will be rendered as plaintext.

ReST provides a wide variety of markup that allows documentation strings to be formatted in a wide variety of ways. Among the many features are different list styles, tables, links, verbatim blocks, etc.

Interpreted text[1] is used to mark up program identifiers, such as the names of variables, functions, classes, and modules. Synopsis will attempt to look them up in its symbol table, and generate suitable cross-references.

[1] http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#interpreted-text

Example C.2. C++ code snippet using ReST-style comments.

```
//. The Nurbs class. It implements a nurbs curve
//. for the given order. It is a very powerful
//. and flexible curve representation. For simpler
//. cases you may prefer to use a `Paths::Bezier` curve.
//.
//. While non-rational curves are not sufficient to represent a circle,
//. this is one of many sets of NURBS control points for an almost \
uniformly
//. parameterized circle:
//.
//. +--+----+-------------+
//. |x |  y | weight      |
//. +==+====+=============+
//. |1 |  0 | 1           |
//. +--+----+-------------+
//. |1 |  1 | `sqrt(2)/2` |
//. +--+----+-------------+
//. |0 |  1 | 1           |
//. +--+----+-------------+
//. |-1|  1 | `sqrt(2)/2` |
//. +--+----+-------------+
//. |-1|  0 | 1           |
//. +--+----+-------------+
//. |-1| -1 | `sqrt(2)/2` |
//. +--+----+-------------+
//. |0 | -1 | 1           |
//. +--+----+-------------+
//. |1 | -1 | `sqrt(2)/2` |
//. +--+----+-------------+
//. |1 |  0 | 1           |
//. +--+----+-------------+
//.
//. The order is three, the knot vector is {0, 0, 0, 1, 1, 2, 2, 3, 3, \
4, 4, 4}.
//. It should be noted that the circle is composed of four quarter \
circles,
//. tied together with double knots. Although double knots in a third \
order NURBS
//. curve would normally result in loss of continuity in the first \
derivative,
//. the control points are positioned in such a way that the first \
derivative is continuous.
//. (From Wikipedia_ )
//.
//. .. _Wikipedia: http://en.wikipedia.org/wiki/NURBS
//.
//. Example::
//.
//.   Nurbs<3> circle;
//.   circle.insert_control_point(0, Vertex(1., 0.), 1.);
```

```
//.   circle.insert_control_point(0, Vertex(1., 1.), sqrt(2.)/2.);
//.   ...
//.
       \
```

To see how this is formatted please refer to the DocBook example[2].

[2] http://synopsis.fresco.org/docs/examples/index.html#docbook