Synopsis Developer's Guide

Stefan Seefeld

Synopsis Developer's Guide

Stefan Seefeld

Version 0.11

Table of Contents

Chapter 1. Introduction

The Synopsis Application Framework is a work in progress. Some of the APIs are quite stable and already used in production. Others are in development. This document wants to provide some guidelines for developers and other adventurers to find their way around the project. APIs that are currently documented here may mature and become stable, at which point their documentation will be migrated to the tutorial[1].

1.1. Origins

The Synopsis Project was founded to support the documentation of the *Fresco* code base, which used a number of different programming languages such as C++, Python, and IDL. The initial design focussed on a high-level, multi-language *Abstract Semantic Graph* which would be manipulated using python *Processor* objects to generate documentation in a variety of formats including postscript and html.

To support multiple languges, Synopsis uses python extension modules to generate a language-neutral ASG. The IDL parser is based on omniORB[2], the original C parser was based on ctool[3], and the C++ parser on OpenC++[4].

1.2. Architecture

Synopsis provides multiple representations of the parsed code, on different levels of granularity. Some of them are exposed using Python, some using C++.

1.2.1. Sub-Projects

Synopsis contains two basic parts: A C++ library, providing an API to parse and analyze C and C++ source files, as well as a Python package to parse and analyze IDL, C, C++, and Python code. While the former provides fine-grained access to the low-level representations such as *Parse Tree* and *Symbol Table*, the latter operates on an *Abstract Semantic Graph*.

Most of the Processorclasses from the Python API are written in pure Python, but some (notably the parser classes) are actual extension modules that use the low-level APIs from the C++ API.

1.2.2. Code Layout

Following the hybrid nature of the project, the source layout has two more or less separate root directories. `Synopsis/` provides the Synopsis Python package, while `src/` contains the sources for the C++ API.

1.3. Current Status: Regression Test Reports

It is important to regularly run tests to keep control of user-visible changes incured by code modifications. The tests that are part of synopsis are not an absolute measure for success or failure, but rather reflect a given state of the system at a particular point in time.

[1] ../Tutorial/index.html
[2] http://omniorb.sf.net
[3] http://ctool.sf.net
[4] http://www.csg.is.titech.ac.jp/~chiba/openc++.html

It sometimes happens that a new addition to the code will make a particular test fail, simply because that test now produces a different output compared to what it used to produce before.

This may indicate a true regression, or it may mean that the *expected output* should be adjusted if the new output is valid and should be the new reference.

The 'official' regression test results are available at all times as a test report[5].

[5] http://synopsis.fresco.org/tests

Chapter 2. The Python API

The Python API in its current form is documented in the Tutorial[1], so only things that are specific to development are mentioned here. As explained in the previous chapter, synopsis was originally designed for code documentation. To support this, An 'Abstract Semantic Graph' representation is used. Only declarations are stored, together with comments preceeding them.

2.1. The Processor Pipeline

One of synopsis' main goals has been flexibility and extensibility with respect to how exactly the parsed data are manipulated. It must be possible for users to define their own output format, or their own way to annotate the source code in comments.

To achieve this flexibility, synopsis defines a *Processor* protocol, which allows multiple processors to be chained into processing pipelines. This way, a user can define his own pipeline, or even define his own processor.

Processors take an ASG as input, and return an ASG as output. One particular processor as Formatters.Dump.Processor, which dumps an ASG into an XML file. This may be useful for debugging purposes.

The scripting language used to define processors in terms of compound processors (i.e. pipelines) is documented here[2].

2.2. The Parsers (Cpp, C, Cxx)

These processors are essentially shared C++ libraries which are loaded and operated by python as extension modules. Their public APIs are discussed in the tutorial[3], and the internals are discussed in the chapter on the C++ API, though some particularities are worth mentioning here.

It is sometimes necessary to debug C/C++ code even when it is controlled through a python API. This makes debugging a bit inconvenient. With gdb, you have to issue **target exec python** before executing the python script that calls an extension module.

2.3. The HTML Formatter(s)

One of the most complex processors is the HTML formatter, which generates html documentation of the parsed source code. There are a multitude of parameters to control many aspects of the formatting. It is documented here[4].

To help to validate the generated set of html pages, synopsis provides **scripts/html-validator**, which can be used to traverse all references between the pages, and to make sure the html is valid.

2.4. Python Regression Tests

The python tests mainly consist in some synopsis script, defining a specific ASG processor pipeline, together with some input in one of the supported languages.

[1] http://synopsis.fresco.org/docs/Tutorial
[2] http://synopsis.fresco.org/docs/Tutorial/pipeline.html
[3] http://synopsis.fresco.org/docs/Tutorial
[4] http://synopsis.fresco.org/docs/Tutorial/html-formatter.html

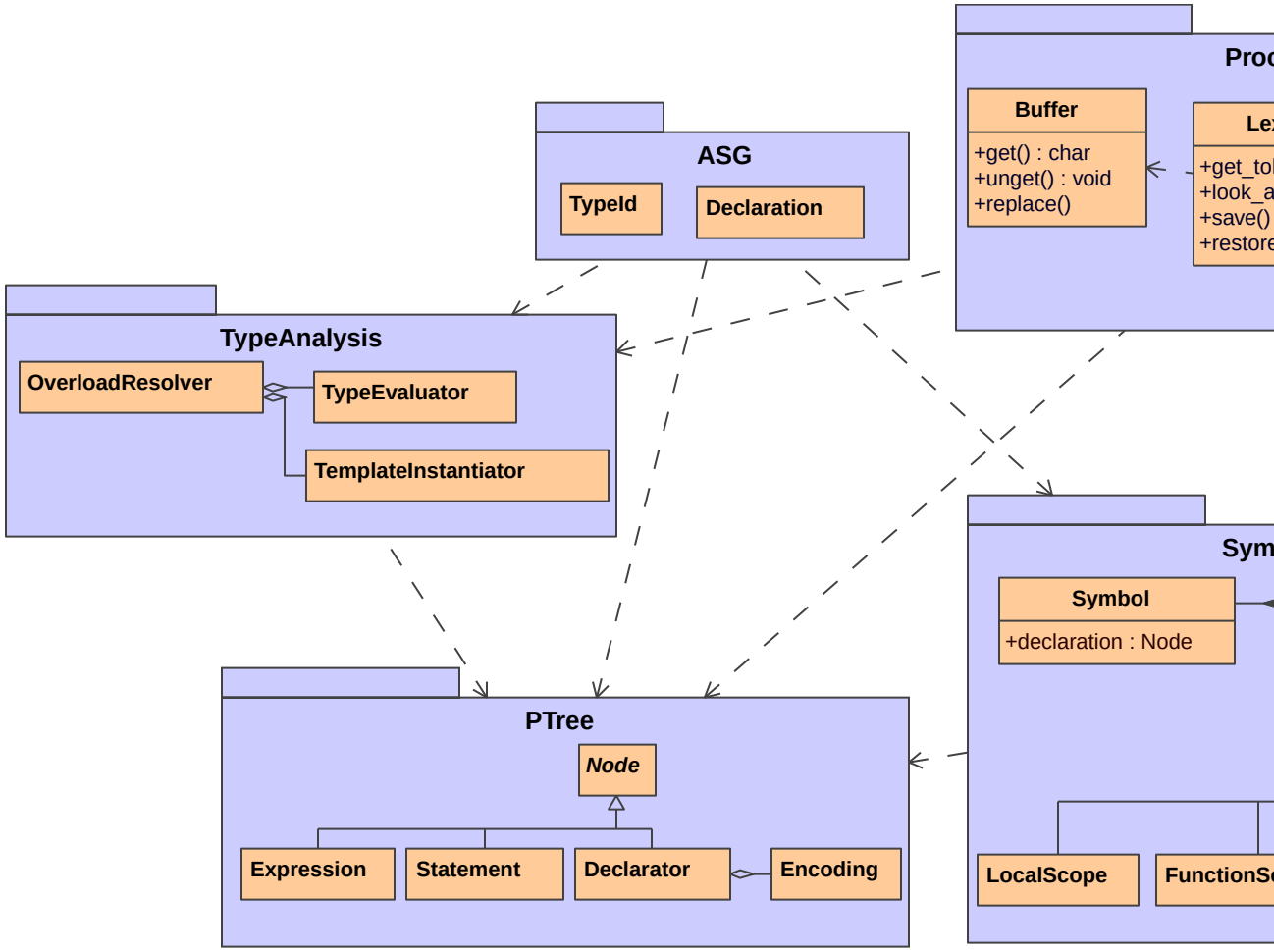
The result will be an ASG that is dumped as *XML* to a file for simple validation.

Chapter 3. The C++ API

The Parser operates on an in-memory buffer of the whole (preprocessed) file. The generated parse tree refers to memory locations in that buffer, and it is possible to replace existing tree nodes by new ones, and then writing the modified buffer back to a file, preserving all but the modified regions.

This feature makes this parser an excellent choice for source-to-source transformation, as the new code doesn't need to be generated from scratch, but instead will preserve all features from the original file that the user didn't explicitely modify.

Parsing a source file involves a number of classes, such as Buffer, Lexer, Parser, and SymbolFactory. These can be constructed with a number of parameters, to control the specific language / language dialect (C, C++, GNU extensions, MSVC extensions, etc.).



3.1. The Parse Tree Module

The parser's principal role is to generate a parse tree. It does that by following language-specific production rules that are followed after encountering lexical tokens that are provided by a lexer.

By means of construction flags it is possible to tell the lexer to accept e.g. 'class' as a keyword (C++) or as an identifier (C). Similarly, it is possible to configure the parser for particular rules.

The parse tree itself is a lisp-like structure. All nodes subclass PTree::Atom (for terminals) or PTree::List (for non-terminals). A Visitor allows to traverse the parse tree based on the real run-time types of the individual nodes (there are about 120 different PTree::Node types).

3.1.1. The Encoding class

The C++ grammar makes it quite hard to recover certain semantic information from syntactic structure. For example, in a simple declaration individual declarators may carry part of the type information for the variables they declare. For example,

```
          char *a, b, c[3];
        \
```

three declarators *a*, *b*, and *c*. The first has type `char *`, the second `char`, the third `char[3]`. In order to avoid the need to analyze the whole declaration to extract the type of a declarator, the parser attaches the type and name to declarators.

A similar argument applies to other cases, where non-local information is encoded into a node's `encoded_name` and `encoded_type` member.

The Encoding class needs to be able to represent full type names, and thus it seems sensible to use a mangling similar (or even identical !) to the one developed as part of the C++ ABI standard (see C++ ABI[1]).

3.1.2. PTree::Display

Parse Trees tend to grow quickly, and it becomes quickly hard to debug them by simply traversing the list. Thus, the `PTree` module provides a simple means to print a (sub-)tree to an output stream.

```
          PTree::display(node, std::cout, false, false);
        \
```

will print the tree referred to by `node` to `std::cout`. The third parameter is a flag indicating whether the encodings should be printed, too. The fourth parameter indicates, whether the actual C++ type of the node being printed should be included in the output.

Since this API turned out to be rather useful, there is a stand-alone applet that just generates a parse tree and then prints it out using the above function.

```
          display-ptree [-g <output>] [-d] [-r] [-e] <input>
        \
```

The available options are:

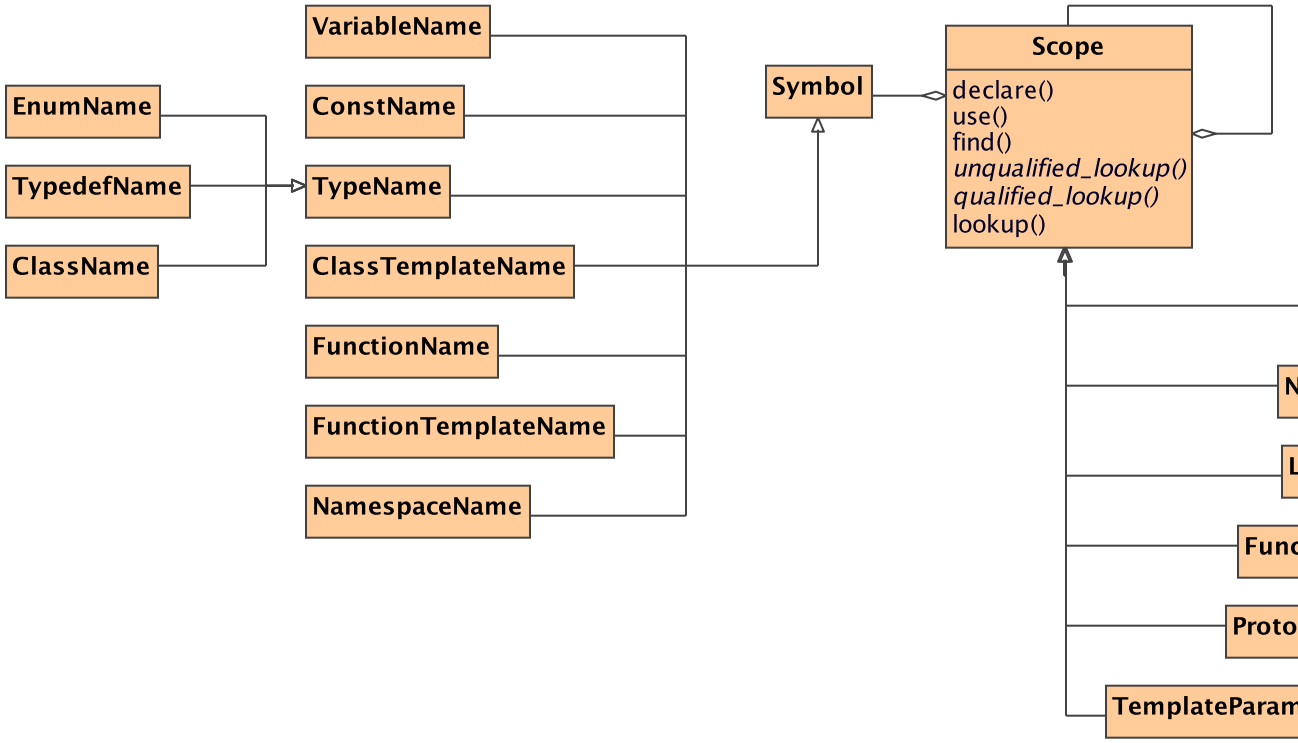
-g `filename`	Generate a *dot* graph and write it to the given file.
-d	Print debug information (in particular traces) during the parsing.
-r	Print the C++ type of the parse tree nodes.
-e	Print encoded names / types for nodes such as names, declarators, etc..

[1] http://www.codesourcery.com/cxx-abi/abi.html#mangling

3.2. The Symbol Table Module

A symbol table is a representation of the source code that provides a means to associate names with symbols, i.e. objects of a specific type. Such a symbol table is a graph composed of interconnected *scopes*, and the type of the scopes as well as the topology of the graph determine how the symbol lookup is performed.

Symbols are declared by the parser as soon as they are encountered, since the parser itself may need them later to disambiguate certain syntactic constructs.



3.2.1. SymbolTable::Display

As a way to inspect the constructed symbol table, the `SymbolTable` module provides a simple API to print the content of a `Scope`.

```
          SymbolTable::display(scope, std::cout);
        \
```

will print the scope referred to by `scope` to `std::cout`.

Again, this API turned out to be rather useful, and so there is a stand-alone applet that just generates a symbol table and then prints it out using the above function.

```
          display-symbols [-d] <input>
        \
```

The available options are:

-d Print debug information (in particular traces) during the parsing.

3.3. The Type Analysis Module

Type analysis is required to a limitted degree during parsing, as well as during semantic analysis that may follow. During *overload resolution* type analysis is needed to find the best conversion (standard or user defined), and for partial template specialization it is needed to match certain types for which a template has to be instantiated.

The type system consists of trees composed of Type nodes which may be looked up in terms of their encoded names (see Section 3.1.1, “The Encoding class”) in a TypeRepository.

3.3.1. The Type Repository

3.3.2. Overload Resolution

3.3.3. The Template Repository

3.3.4. Type Evaluation

3.3.5. Constant expressions

3.4. C++ Regression Tests

The main set of C++ tests currently performed by the regression test suite is concerned with symbol lookup. Individual tests are copies of the code from the C++ specification, mainly clause 3.4.

Most of the failing tests fail because they haven't been implemented yet, i.e. there isn't even some *expected output* to compare against. As the SymbolTable module is completed, these tests should eventually all be *passed*

Further, more tests should be added that cover other aspects of the parser, such as type analysis.