

The threaded Flex/Bison API pattern of pysfst

Toni Arnold, Zurich

2006

Abstract

The development of the pysfst Python interface for the SFST finite state transducer (a flex/Bison application) using SWIG led to a generalizable technique for building interfaces to specialised domain languages.

Usually, these compilers operate on the filesystem, use stderr for error messages and generate binary files which are interpreted by a special application. But small compilers integrated into other languages often operate completely in-memory. An example is Python's module `re` which compiles simple strings into regular expression objects. The pysfst application tries to bridge that gap, converting files and compiler errors of the domain language into pipes and exceptions in the target language.

1 The software environment

Any software involved in the pysfst project is Free Software.

The pysfst module contains Python bindings for the API and some executables of SFST, the Stuttgart Finite State Transducer Tools. The interface was generated with SWIG, but uses a lot of glue code both in C++ and Python. SFST itself is written in C and C++. Its domain language parser is generated using flex and Bison.

This article was written with the Kile editor and LaTeX on a Gentoo Linux box. The uml diagrams were drawn with dia.

2 From console applications to objects

A minimal shell session for compiling and using a trivial transducer with the SFST might look like this:

```
$ echo "spam" > spam.fst
$ fst-compiler spam.fst spam.a
spam.fst: 2

$ echo "spam" | fst-infl spam.a
reading transducer from file "spam.a"...
finished.
> spam
spam
```

The source file “spam.fst” is compiled to the binary transducer “spam.a” which in turn is interpreted with the `fst-infl` application. The transducer recognizes the input string “spam” and writes the analysis to stdout: “spam”.

The same effect can be achieved with this Python one-liner using pysfst:

```
>>> sfst.compile('spam').analyze('spam')
['spam']
```

The sfst compiler generates a Transducer object which provides the method analyze() which returns the analysis in a list.

2.1 The hidden common data flow structure

Both the stand-alone and the embedded compiler share a common data flow structure. There is some source code which can be compiled into a binary form. This binary is executed by an interpreter. The source code allows include statements which can refer to other source code or to pre-built binaries for combining with the main source in a later build step.

Therefore there are basically four fundamental data types involved: compiler source code, resulting binary code, interpreter input and interpreter output. In the SFST case, only the binary code is of non-textual form, but the interpreter input/output could also be a machine readable binary. Figure 1 illustrates this in a data flow diagram.

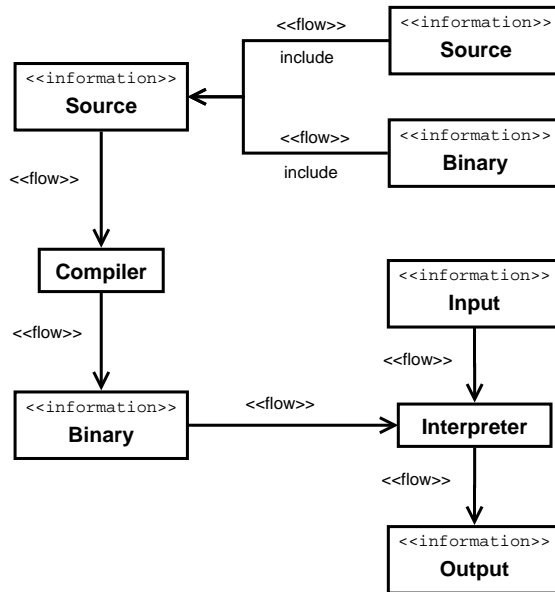


Figure 1: Compiler/Interpreter data flow

3 Interface depth

The following subsections of 3 introduce different interface types in an incrementally deepening order.

3.1 Filesystem with processes

In the stand-alone version, every data flow necessarily involves a filesystem access, except the interpreter input/output flow which can run e.g. in an interactive shell using `readline` if it is textual. The direct way to embed such a compiler environment into another language is to use the filesystem for any data flow involved. Any method call to the interface forks a new process and starts the compiler or the interpreter with the corresponding file paths.

The `XFST::Lookup` module described in⁷ for the proprietary Xerox Finite State Tools provides for example such an interface for Perl. There is no interface to the compiler itself, and the `Lookup` constructor takes a filesystem path to the compiled binary as argument:

```
# to use a single FST:
my %looked_up_wordforms = $lookup->lookup({
    fst => 'path/to_a/finite_state_transducer',
    words => \@wordforms_as_array_REFERENCE
} );
```

3.2 Interpreter library

But this is not what people normally expect of a real interface, because forking a new process for each method call is costly. Consequently, above `lookup()` method takes an array of words as argument. The first step is to replace at least the interpreter input/output flow with object method calls. The interpreter itself is still instantiated with a file path to the binary, but it is loaded as a shared library.

If there's no access to the source code of the interpreter, usually two shared objects are needed: the shipped interpreter binary and the glue code library, either written manually or automatically generated e.g. with SWIG. The glue code translates target language method calls to C++ method calls.

In the SFST case, the interpreter is implemented as a C++ class, the `Transducer`. Because SWIG supports C FILE objects, the constructor is not a file path, but a target language file object. A sample session thus would look like this:

```
>>> file = open('spam.a', 'rb')
>>> transducer = sfst.Transducer(file)
>>> file.close()
>>> transducer.analyze('spam')
['spam']
```

3.3 Interpreter objects with operators

In the SFST case, the interpreter is implemented as a C++ class, the `Transducer`. This class supports various unary and binary operators for manipulating and combining several interpreters. All supported operators are described in the `operators.txt` doctest. This is a sample session for the disjunction operator “|”, here with directly instantiated transducers (the required alphabet definition was omitted):

```
>>> fish = sfst.Transducer('fish')
>>> bird = sfst.Transducer('bird')
>>> fish_or_bird = fish | bird
```

```
>>> fish_or_bird.analyze('fish')
['fish']
>>> fish_or_bird.analyze('bird')
['bird']
```

3.4 Compiler with the filesystem

Of course it would be relatively easy to add an interface to the compiler without spawning a new process for each compiler run. An indispensable precondition for such an intention of course is access to the source code, because it needs to be recompiled with an entry point for the target language, bypassing the application's `main()` routine.

In the flex/Bison world, parsers normally are not thread safe unless they are explicitly generated as pure parsers with the `%pure-parser` declaration. Of course this doesn't make the whole compiler thread safe unless all the custom code called by flex/Bison is thread safe, too. For console applications, this is rather unlikely, so the compiler call has to be guarded with mutex semaphores like in this simplified example which uses POSIX threads:

```
#include <pthread.h>
pthread_mutex_t compiler_mutex =
    PTHREAD_MUTEX_INITIALIZER;

extern void compile(...)
{
    if ( pthread_mutex_lock( &compiler_mutex ) )
        warn("fst-compiler.compile:_pthread_mutex_lock_
            failed");
    ...
    yyparse();
    ...
    if ( pthread_mutex_unlock( &compiler_mutex ) )
        warn("fst-compiler.compile:_pthread_mutex_unlock_
            failed");
}
```

This low-level implementation follows the metaphor of using thread semaphores directly where the problems occur and therefore exclusive access is required: immediately around the `yyparse()` function. But it has the disadvantage that this solution is not portable. On Windows Python uses its specific threading mechanism, and `<pthread.h>` is not even present. A portable metaphor is therefore to set thread semaphores where threads can be spawned, and this is Python. Therefore `pysfst` switched to Python semaphores with version 1.1.3:

```
import threading
compile_lock = threading.Lock()
def __swig_compile(*args, **kwargs):
    compile_lock.acquire()
    try:
        __swig_compile_direct(*args, **kwargs)
    finally:
        compile_lock.release()
```

3.5 Compiler with pipes

The tightest interface - without replacing the entire file i/o with custom buffers all over the original application - is to use pipes for the source code and the resulting binary. The `pysfst` interface code itself encapsulates the low level code for creating, writing to and closing the pipes. The Python code for creating a pipe read/write pair is rather simple:

```
def pipe_text_file():
    (fd_read, fd_write) = os.pipe()
    file_read = os.fdopen(fd_read, 'r')
    file_write = os.fdopen(fd_write, 'w')
    return (file_read, file_write)
```

At least on Linux 2.6, there is a severe limit on this approach: The kernel buffers only 64KB, see `pysfst's` `pipe_limit.txt` doctest for details. If there is more data to pass through the pipe, the data producer must run in another thread than the data consumer. When put this way, the mechanism resembles to a Unix shell pipeline. The simplified Python code for the `Producer` class with threading contains a method for starting the thread:

```
class Producer(Future):
    def start(self, file_write, target, args, kwargs):
        thread_write = threading.Thread(
            target=self.exec_target,
            args=(file_write, target, args, kwargs) )
        thread_write.start()
```

3.6 Compiler with target language exceptions

However, it is still uncommon to have a `stderr` pipe to parse for error messages if there are e.g. syntax errors in the source code. Therefore the compiler should throw an exception – or whatever the default error handling in the target language is – on any errors it normally would write to `stderr`.

This poses no real problem as long as the producer of the exceptions doesn't run in another thread than the consumer of the expected output. If there is a pipeline where the first producer thread throws an exception, the next thread will get incomplete data and throw an exception, too. Because the call stack of the main thread is different from the producer thread, the user will get an exception like “wrong file format” instead of the original “syntax error”.

The `ExceptionDependency` base class encapsulates the exception handling tactics: Any class depending on the output of another class takes the dependency class as constructor argument. An exception occurring somewhere in the pipeline is immediately caught and privately stored. The method directly called by the user of the library must call the `raise_exception()` method which recursively walks up the dependency chain until the first `Producer` class which depends on nothing. Then it throws the first exception it finds. With this technique, the logically primary exception is thrown instead of the last one. Here's the code of the `ExceptionDependency` class:

```
class ExceptionDependency(object):
    def __init__(self, dependency):
```

```

        self.exception = None
        self.dependency = dependency
    def raise_exception(self):
        if self.dependency != None:
            self.dependency.raise_exception()
        if self.exception != None:
            raise self.exception

```

3.7 Compiler with includes of target language objects

There's still something left: It is common to split huge source code files into smaller ones and then combining them with the `#include` statement. The SFST furthermore allows including pre-built binaries. Of course such builds usually are done with a makefile or the like, but there are situations where a compilation using includes from the target language makes sense, e.g. for adding a single word to a lexicon in a spell check session.

Technically, this involves a callback from the flex scanner to the target language for reading the included data. In SWIG, the classes for handling callbacks are called director classes. All virtual methods of their base class are extended such that a C++ method call ends up in a corresponding method call in the target language. The C++ code for opening a file with the `Include` class looks like this:

```

void Include::open( char *path, char *mode )
    throw(char const*)
{
    this->incr_file_ptr();
    this->set_file(fopen(path, mode));
}

```

However, this code is only called when the flex code is compiled as part of the console compiler. But the `pysfst` interface provides a class inheriting from the `Include` wrapper class generated by SWIG. Here's a simplified excerpt, showing the part when a source file is to be included from a python string provided by the `include_dictionary` instead of the filesystem:

```

class IncludeDirector(Include, ExceptionDependency):
    def open(self, path, mode):
        file_content = self.include_dictionary[path]
        (file_read, file_write) = pipe_text_file()
        producer = Producer(None)
        producer.start(self, file_write,
                      file_write.write,
                      (file_content, ), {})
        self.set_file(file_read)

```

The crucial point is the `set_file` accessor: It is inherited from the C++ base class above and handles – via the SWIG glue code – the conversion from a python file to a C file descriptor. From the viewpoint of flex, there is no difference between a file opened from the filesystem in C or a read end of a pipe opened in Python.

4 Putting it all together

4.1 The class diagram

The class diagram in Figure 2 gives a detailed overview of the reusable classes used in the pysfst interface. The method's argument lists were simplified for better readability.

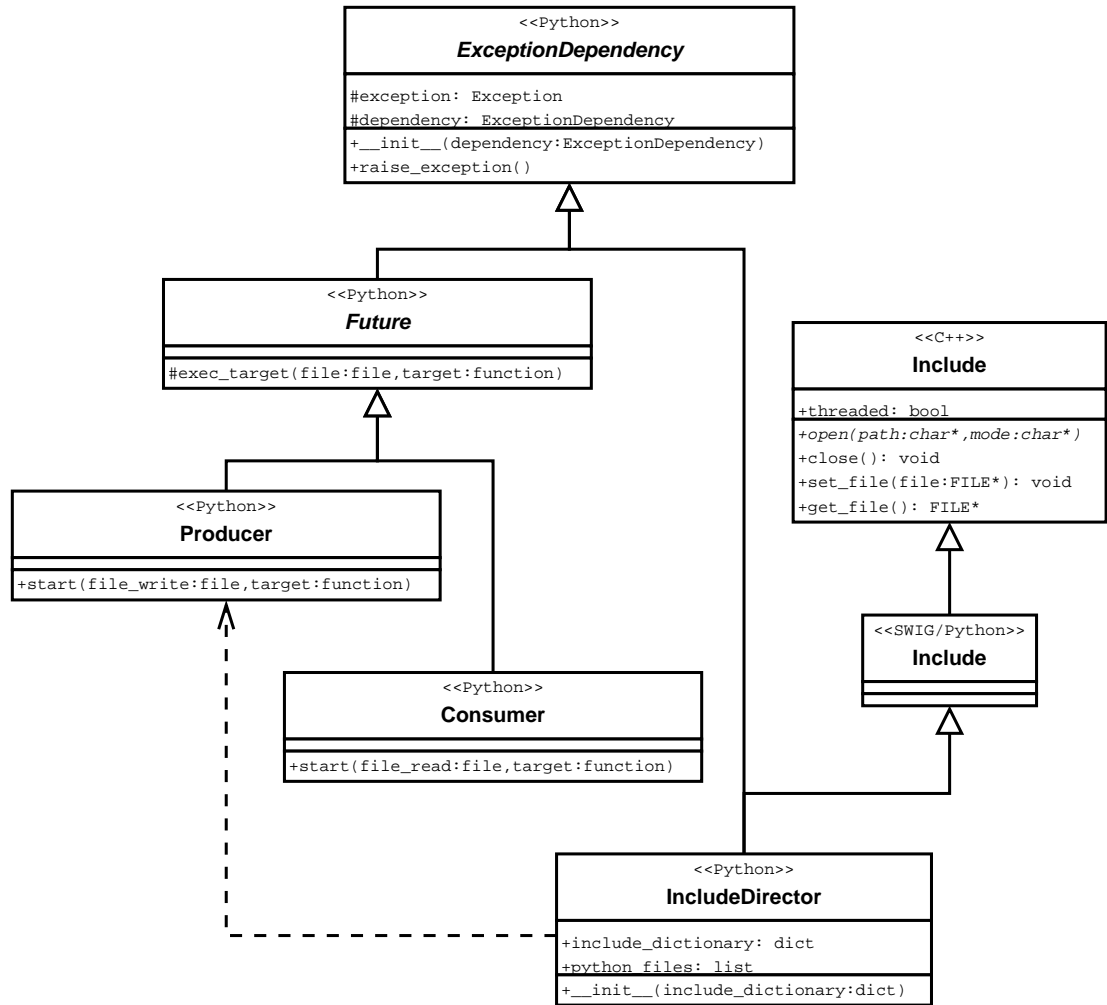


Figure 2: Classes used in the pattern

4.2 The threading mechanism

4.2.1 Non-Threaded

The simplest case is a non-threaded instantiation of an interpreter class, in the SFST case a Transducer object. It needs a binary transducer which is just a python string containing the code. The Producer writes all data to the write end of the pipe, the Transducer constructor blocks until the Producer has finished, then it starts the Consumer to

read all data from the pipe. The method passed to the Consumer is the constructor of the Transducer taking a file as argument. Figure 3 shows the the two non-overlapping start method calls.

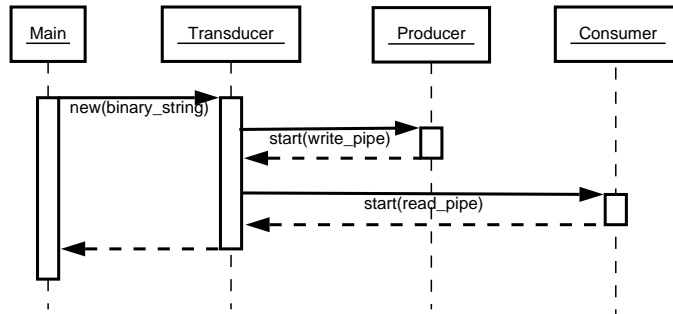


Figure 3: A non-threaded Producer/Consumer pair

4.2.2 Threaded

In the threaded case, the start method actually spawns a new thread. The Producer writes to the pipe until the buffer is full and it blocks. The Consumer keeps reading on the read end of the pipe until EOF. Therefore the activities of the corresponding Producer/Consumer methods overlap now. Figure 4 shows the the two overlapping start method calls: The Consumer is called while the producer did not yet return.

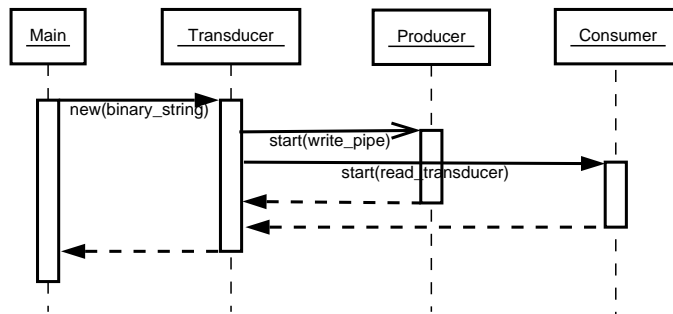


Figure 4: A threaded Producer/Consumer pair

4.2.3 Threaded Exceptions

Any class inheriting from ExceptionDependency can pass a dependency to the constructor. If an exception occurs in the Producer (the dependency), it is immediately caught, locally stored in an attribute and the thread terminates.

Afterwards, there will also be a follower exception due to bad input in the Consumer (the dependent), which will be caught, too. At the end, the Consumer calls its `raise_exception()` method, which recursively is passed up the dependency chain to throw the original exception.

Figure 5 show the same objects and method calls as Figure 4, but the Producer

method throws a caught and stored exception which will be raised afterwards. Exceptions are indicated with the non-standard exception stereotype.

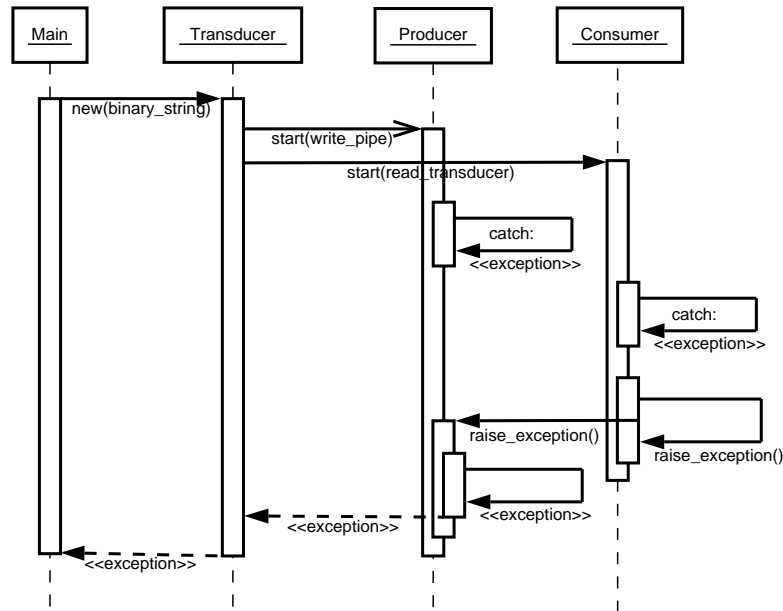


Figure 5: A threaded exception

References

1. Bison. A general-purpose parser generator. URL <http://www.gnu.org/software/bison/>.
2. dia. A diagram creation tool that supports uml modeling. URL <http://www.gnome.org/projects/dia/>.
3. flex. The fast lexical analyzer. URL <http://www.gnu.org/software/flex/>.
4. Gentoo. A special flavor of linux that can be automatically optimized and customized for just about any application or need. URL <http://www.gentoo.org>.
5. Kile. An integrated latex environment. URL <http://kile.sourceforge.net>.
6. LaTeX. A document preparation system. URL <http://www.latex-project.org>.
7. Alex Linke, Rona Linke, and Bjrn Wilmsmann. Hinweise zur installation und verwendung des constraint-taggers, 2005. URL http://www.topicalizer.com/files/tagger/Dokumentation/tagger_doku.pdf.

8. operators.txt. Infix operators for the “transducer” class. URL <http://home.gna.org/pysfst/tests/operators.txt>.
9. pipe limit.txt. The 64k pipe buffer limit. URL <http://home.gna.org/pysfst/tests/pipe-limit.txt>.
10. pysfst. Python bindings for the sfst library. URL <http://home.gna.org/pysfst/>.
11. Python. The python programming language. URL <http://www.python.org>.
12. SFST. Stuttgart finite state transducer tools. URL <http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html>.
13. SWIG. Simplified wrapper and interface generator. URL <http://www.swig.org>.