The Aditi story

Or, how I learned to stop worrying and love templates and documentation.

Overview

- A case study of using templating and structured documentation in the Aditi project.
- Using scripting to generate html, python, C and C++.

How a simple tool ended up being amazingly useful all through the project.

Who am I?

Cameron Blackwood, research programmer on the Aditi project at the University of Melbourne. The Aditi project consists of Liam Routt programming the core engine and myself working on the network API.

Aditi Project

Aditi is a deductive database. Was ground breaking 10 years ago, but other databases have caught up.

- Has very complex relation schema (table definitions) that can be functional and recursive.
- Supports injecting smart requests into the server to decrease the amount of work the engine has to do. These are generated with the Mercury language.

Mercury Language

Mercury is a logic constraint language. Mercury generates the constraint searches that Aditi uses. Constraints limit help reduce database fetches.

For example: flight planning. If you have all available flights in a database, then the smart request can use constraints to limit the data pulled from the database by ruling out certain options. For example, dont pull out flights from the past, ignore places you have already been, dont use airlines you wont fly on, etc.

The problem

Arriving at the project to get the network API working, it was quickly decided to redesign and totally rewrite the API.

The existing API had a number of problems:

- it was undocumented
- it used parts of the engine internals without a clean API and the engine had changed since it was written
- it used an RPC library which had problems with long requests in that it would kill the 'idle' connection.

Design Phase

Design goals for the new api were:

- a clean interface with the engine, rather than using random internal functions
- a clean interface with the client, naturally as transparent as possible
- implement the required core functionality only
- document it well

Given that an engine/API interface would be defined and documented, it seemed logical to make the network API provided to the client as clean and close to the engine interface as possible.

If that was the case, then the API layer needed to perform the following steps for an api call.

1) offer a function which takes arguments
2) convert the arguments into a data stream
3) send them to the server/engine part of the API
4) decode the data stream back into arguments
5) call the engine version of the required call

The document describing the API has most of the information required to build most of those steps. Looking at those steps again, the design document:

1) details the API function
2) provides the type and structure of the data we need to encode
3) does not help here
4) provides the details of how to decode the stream
5) tells us about the engine version of the function

Most of that is based on the API document itself, thus if we have a structured document, it may be possible to generate most of the API code itself via the document and a template.

```
<relation_tuple_add>
====================
Comments:
---------
  Adds an entry to a relation.

  Tuple data of "" (i.e. an empty string) may be used with NULL-arity relations
(tuple data should be ignored in such situations, but some string needs to be
passed anyway).

Inputs:
-------
  apiID:session_ticket: session ticket
  apiID:transaction_ticket: transaction ticket
  apiID:relation_ticket: Relation ticket for relation to empty
  apiSTRING:tuple_data: string encoding the data to use.
  apiBOOLEAN:use_internal_save_point: an internal setting.

Outputs:
--------
  uint64:tuple_fps:     Future file/page/slot for the new tuple

Return Values:
--------------
  AS(OK) = worked
  another error = fail
```

Sample API documentation for an API function

The API doc and generated code

The key parts of the API covered in the document are the API functions offered by the engine. The document specifies: function name, inputs, outputs, return values and comments for each function. Inputs and outputs are themselves further detailed with a name, type and comment.

The python parser for the API document, created a list of functions, each with a list of inputs and outputs. Each of these things, functions, inputs and outputs, had a number of static variables which mapped the setting to useful values.

For example, an input defined variables such as: the api type, the engine type, how to convert from one to the other and what the 'zero' value of the variable is. So, given that kind of information, a templating script was created that used that information to expand template files and fill in this information.

The first autogenerated code was a simple 'direct' pass through connection from the client side API directly to the Aditi engine interface. It did only minimal translation and had no network interface at all.

So, to generate 'direct' interface code, we need to have a template which would perform the steps outlined in “Skeleton of an API wrapper function”. Each step is reasonably simple and easy to generate, given the information about the function and its types. So a few lines of template code can generate the connection between the client and the engine.

- generate function prototype
- convert C inputs to C++ arguments
- call engine with C++ arguments
- convert C++ results into C
- return the C values

Text 1Skeleton of an API wrapper function

```
  /* Declare and convert engine inputs */
[[for args.in ]]  %(t.etype)-15s   %(a.name_eng)s = %(a.api_to_eng)s;
[[/for args.in]]
  /* Engine output defs */
[[for args.out]]  %(t.etype)-15s   %(a.name_eng)s = %(t.zero)s;
[[/for args.out]]
```

```
  /* Declare and convert engine inputs */
  GenericId  ENG_session_ticket = ticket_decode( API_session_ticket );
  GenericId  ENG_transaction_ticket = ticket_decode( API_transaction_ticket );
  GenericId  ENG_relation_ticket = ticket_decode( API_relation_ticket );
  LocalString ENG_tuple_data = API_tuple_data;
  Bool       ENG_use_internal_save_point = API_use_internal_save_point;

  /* Engine output defs */
  uint64            ENG_tuple_fps = 0;
```

Convert C inputs to C++ arguments

Results:

+ generated lots of code (over 50 functions).
+ was very quick and easy to do
- didnt have a lot of trust in it and spent a lot of time verifying it
- needed exceptions for the few functions which were in the client API that were not mirrored in the Engine interface. For example: the client 'connect' call is not in the Engine interface

- had some templating problems (discussed below)

Templating problems

Templating had some problems, such as whitespace formatting to make the code more human readable and joining argument lists together neatly.

You cant just put a comma after each argument because the last argument does not need one, so you have to have specific code to do that. It was a little ugly, but much easier than hand writing 50 functions.

Interconnection between the API and the engine.

The 'glue' between the API and the engine has to perform the following tasks: encode the arguments, send them, wait, receive the results,decode the results.

The first version, discussed above, was a direct transport with pretty well nothing in the middle. It took the arguments and the passed them almost directly to the engine.

By adding some 'layers of glue' in the middle it was quickly possible to build up and test a more fully functioning interconnection.

The next layer coded was one that took the arguments and put them into a C list of variables. This list was passed to the engine side where the list was unpacked and turned back into arguments to use with the engine. (The results came back via the same kind of path.)

The next layer was one that took the C list of variables and added to that a layer that encoded the list into a portable 'string' format. This string was passed to the engine side which converted it back into a C list and extracted the arguments from that list.

Both of these layers were really small because the information of how to pack the C list was defined in the API document and thus could be automated, again via the use of templates.

The goal here was a string transport, because a string can be used in a number of ways to send requests between separate processes.

The first real use of the string layer was to log a 'history' of strings into a file. Once there was a trace file, we could create a 'server' which would read the strings from the file and 're-execute' them.

A network server could be created at this point, except that it would require code to track and use network connections, so instead a more simple approach was taken.

Python interfaces

Given we how had layers that seperated the client from the server (via the encoded strings) we now started to need testing scripts to verify the correct operation of the database. Because our only interface was the C API, that pushed us towards using C for our test scripts. C is not the greatest language to handle tests such as 'load a set of data strings from a file'.

Python is a lot better for scripting and we already required python for the template language, it was decided to look at generating a python interface for Aditi.

Initially we looked at SWIG which seemed to offer the ability to pass a header file and to generate a python module, but our project required long 64 bit integers and SWIG had weak support for them. It could support them if you wrote some helper code, but if we have to write code, why not use the templating system to generate the code?

The outline of the approach for each function was quite simple and is outlined in "Skeleton of a python interface function". It was quite similar to the client API / Engine interface code, so a template was created and more variables were added to the API loader and template code.

Alas there were a few differences between the C API and the python API which required hand coding and additional exceptions, but most of the functions were automated.

The differences were mostly caused by the different ways that C and python handle variables, for example
 * C uses: int argv,char *argc[]
 * python uses: sys.argv

Overall the result was a very easy and useful python module, again generated almost totally from the API document.

```
* take args
  - get input as python object
  - turn python object into C var

* call engine(...)

* make return list
  - turn output C var into python
object
  - put python object in return list
```

Skeleton of a python interface function

Python and Strings

Python is excellent at short scripts which process strings the string format of our requests allowed us to pass requests and results around quite easily.

A 'replay' server (to rerun string logs) was written in python for about 5k of code.

A network server was written in python for 7k of network code and 5k of Aditi API code. This server was able to receive and spool data to and from multiple clients, allowing one engine/server to serve many clients.

Clients (written in C) also had to have network code, but it didnt only had to track one connection and was much simpler.

Debugging output was added to the 'string' layer to allow logging requests to a file and the 'replay' server was modified to provide a general debugging tool. IE, turn on debugging, write the strings to a file and you can use that in a problem report.

A better python wrapper

Python is a object oriented language. C is a function-based language. Using the python interface (which

mapped directly to the C API) was very much like writing C and had none of the advantages of object oriented programming. What was needed to make scripting easier was a more python like interface where, for example, the relation object knows which transactionId and sessionId to use when calling the C API.

C:
relation_tuple_add(sessionId, transactionId, relationId,data,flag)
python C interface:
module.relation_tuple_add(sessionId, transactionId, relationId,data,flag)
python with objects:
relation.tuple_add(data,flag)

We created wrappers around the raw interface to create objects for our various database objects. The interface was much cleaner and much easier to use, but it needed to be hand written as it was feasable to generate this interface from the API document.

Lessons from an API change

Work on the database and mercury (the client) required that the API be changed. Because of the autogeneration, the required steps to modify the network API library were simply to edit the document and rebuild the interface.

There was a lot of checking and rechecking the interface, because of the lack of human involvement, but the interface worked perfectly after the rebuild. The change also rebuilt the python C interface module and added the new call, making it clear that autogeneration was a successful and powerful tool.

Client side API helpers illustrate a problem

With testing taking off, it became clear that the 'core' interface to the engine required similar blocks of code to initialise many of the tests. The decision was made to try and offer some 'helper' functions to clients.

For example, almost all test code had to load a relation into the database for testing. This was a long task, even with the python wrapper, as it took multiple steps to create the relation, load any indexes, load the data and then close and commit. It was decided that offering an API call to do that in one step was a better solution, although we did not want to add such a function to the full API.

The function was written in the client part of the code and the API document was updated and a rebuild started. The rebuild failed because the new functions were not offered by the engine, thus the engine side of the interface failed to build. Again, exceptions to the templating had to be added.

Exceptions had been required for each change since the start so it was becoming obvious that some better solution was required to this problem.

API document updated

The exception problem was solved by adding a 'flags' section to each function in the API which would tell the template when to build which functions.

This small change moved the decision of where and when to

```
Flags:
------
  engine    = yes
  core      = yes
  api       = yes
  pythonapi = yes
```

Additional Flags section of API documentation

build things from the template script, into the function information (where it belonged). The template itself could now turn it on and off, rather than requiring specific exceptions in the script itself.

```
[[for functions]]
[[if f.gen_engine]]

/* ---------------- begin of function %(f.name)s ---------------- */
...
/* ---------------- end of function %(f.name)s ---------------- */

[[/if f.gen_engine]]
[[ifnot f.gen_engine]]
/* ---------------- Skipping function: %(f.name)s ---------------- */
[[/ifnot f.gen_engine]]
[[/for functions]]
```

Template use of API flags

Ease of making changes

An issue with the Mercury interface highlighted the lack of checking for NULL pointers passed for the output pointer.

To fix this, a small template update was required in a number of template files to check each output to see if it was NULL and to return an error if it was.

Once again, autogeneration made this simple. It required only adding a small piece of code to a few templates. But note that this output was generated for every output on every function. A small code change to a few templates had updated all our code automatically.

Fixing Templates

Fixing the exception issue had illustrated problems in the template system itself.

It had been developed to just generate the API stub, but already that had expanded to html documentation, python interface and helper information for the 'glue'. It was difficult to get the output file whitespace to look correct and it often required a lot of code to do reasonably simple things, such as place commas in the correct place.

Most of these issues should have been obvious, as I had used the Albatross cgi toolkit previously and I had similar issues which Albatross provided solutions for. It was hardly a surprise that issues Albatross template developers had struck would also effect Aditi templates. So, it was decided to use the templating engine from Albatross in Aditi.

Albatross is a web cgi system written in pure python that has an elegant design. It creates a 'context' object with a name space, which you load with your python objects. Taking your template and passing it and the context to the template engine generates your output. Notice in the example “Albatross template example” we are using a method 'printf' on the object a (the output argument).
Albatross provided:

```
<al-for iter="oarg" expr="f.outputs"><al-exec expr="a=oarg.value()">
robj=<al-value
      expr="a.printf('%(ptype_c2p_fun)s(%(ptype_c2p_arg)s C_%(name_api)s );')"
      noescape>
if ( NULL==robj ) { return NULL; }
if ( 0!=PyTuple_SetItem(rv,  anum++ , robj ) ) { return NULL; }
</al-for>


robj=PyLong_FromUnsignedLongLong( C_API_relation_ticket );
if ( NULL==robj ) { return NULL; }
if ( 0!=PyTuple_SetItem(rv,  anum++ , robj ) ) { return NULL; }
```

Albatross template example

+ more advanced whitespace processing
+ a much cleaner interface
+ ability to use objects & methods

but at a cost of:

- html escapes were a problem because Albatross was designed for web use and thus escaped its output quite often. It provided a noescape tag to turn off this behaviour.
- for loops in Albatross return iterators, not values. Iterators have a number and a value, and thus one needs to use .value() to extract the 'item' from the iterator

More template uses

Changes to Mercury required the use of dynamic link libraries and given that we now required them, it seemed logical to create an Aditi dll to make development easier. Aditi generates a 50 Megabyte library to link against which makes recompiling the client or API library quite slow. The ability to use an Aditi dll would mean small clients which retained the ability to directly use the engine.

The template method was easily used to generate the library stubs and the library itself, as noted previously, most of the information was present in the document.

Yet more template uses

Locating memory leaks in the client code was quite easy, as one could simply use a network transport (without an inbuilt engine) and trace that program.

Locating leaks in the server side of the API was much harder because of the size and complexity of Aditi. Aditi would confused valgrind (our leak locating tool) and make tracing leaks in the network server difficult.

The solution was to generate a 'fake' Aditi engine, using the API document and templates. The fake engine would just return a random value of the correct type (as defined in the document) but not actually do any other work or require the Aditi engine. Without the Aditi database to confuse valgrind and with such a simple 'engine', locating memory leaks in the server side of the API was easy. A bonus was the ability to use the 'fake' engine for performance testing of the API.

Conclusion

This, at the end of the project, we had:

- a client C interface
- a python interface
- interface to the Aditi C++ engine
- python object wrappers
- a dynamic aditi library
- a fake 'testing' engine

By creating a structured API document and a template system all but one (the python object wrapper) of these was mostly automatically generated. Also, because of the large amount of automatically generated code, the API was very easy to maintain and update (usually just edit, build). Another bonus was that the documentation always correctly reflected the interface.

More than 50% of the final network code was generated from the templates. Notice that the netcode total is: 100k python, 300k of C (of which 90k is testing), 60k of template and yet it generates 570k of code, vrs 460k of human code.

```
                  files     lines       bytes
-----------------------------------------
aditi  from cvs  2,058   505,334  16,845,729 !CVS
         python    102    12,203     341,510   '\.py$'
        h/c/c++    817   194,872   5,097,890   '\.[ch][ch]$'
      templates     19     2,198      62,333   'TEMPLATE'
code total=========================5,501,733==

API    from cvs    165    21,471     819,710 !CVS & 'netapi*'
         python     22     3,711     110,830   '\.py$'
        h/c/c++     59    11,207     303,003   '\.[ch][ch]*$'
   test h/c/c++     15     3,036      89,958      + 'test'
      templates     15     2,115      59,987   'TEMPLATE'
     code total======================473,820==

templates           19     2,198      62,333 'TEMPLATE'

generated           16    23,126     573,669 'GENERATED'
```

Code size in aditi

The following really worked in the project:

- it was driven by a simple text editor
- it was very easy to change
- it just worked
- it created a great trust in automatically generated code

Problems that arose:

- the problem with exception was annoying and should have been clear as soon as the project started to reuse the templating system
- the need for joins and hacks in the template system should have more quickly illustrated the foolishness of reinventing that wheel

References:

- Albatross - A toolkit for stateful web applications

http://www.object-craft.com.au/projects/albatross/

- Mercury - A logic/functional programming language
  http://www.cs.mu.oz.au/research/mercury/
- Aditi - A deductive database
  http://www.cs.mu.oz.au/research/aditi/
- Valgrind - A debugging and profiling tool
  http://valgrind.org/
- Python - An interpreted, interactive, object-oriented language.
  http://www.python.org/