WATT: A Compiler for Automated Visualization Service Generation

Evan F. Bollig$^1$, Martin D. Lyness$^2$, Gordon Erlebacher$^1$ and David A. Yuen$^2$

GCE ’07
11 November 2007
Reno, NV

$^1$ School of Computational Science, Florida State University
$^2$ Minnesota Supercomputing Institute, University of Minnesota
Background

- Large consortia investigate specialized topics with similar problems:
  - Task automation (workflows, load balancers, compilers, etc.)
  - Public access to utilities (repositories, portals, etc.)
  - Distributed computation

- In VLab (http://vlab.msi.umn.edu) everything should be a service
  - Compute services
  - Task management services
  - Visualization services
Can we automate service creation?

- Process to write a visualization service:
  1. Write VTK (Tcl) script to render example data
  2. Convert Tcl to C++ (by hand)
  3. Add service details (SOAP, WSDL, security, etc.)
  4. Compile and deploy

(Repeat)

- 2 and 4 stay the same; 3 rarely changes
The Web Automation and Translation Toolkit (WATT)

- “Compiler” (> Translator)
  - Tcl to C++
    - Requires type inference
  - Adds gSOAP to C++
    - gSOAP provides SOAP handlers, WSDL, stub and skeleton
    - WATT connects gSOAP to core code
      - Code Generation

- Developers write Tcl script, WATT does the rest. (Almost...
WATT Execution

Diagram:

- Tcl Scripts
- WATT Compiler
- gSOAP Classes (.h/.cpp)
- Visualization Web Service Classes (.h/.cpp)
- gSOAP Compiler (soapcpp2)
- VTK C++ Implementations (.h/.cpp)
Defining the Stub

- **Standard set of published methods**
  - Allow clients calling standard methods to work with any service
    - `renderBase64Binary` returns image as byte array
    - `jumpToView` move camera to view all rendered objects along specified vector
    - and more…

- **Tcl “proc” commands**
  - Expand functionality of service
    - Can be associated with UI toggles, value modifiers, etc.
Determining Types

- Register system provides type mapping hints
- "proc" types determined by last statement and parameter usage
- Two phase type inference:
  - Generate abstract syntax tree and match known types with registers
  - Walk tree again assigning types based on other usages
Input Limitations

- **Limited Tcl syntax**
  - No variables in strings (i.e. “hello $name”)
  - No variables in names (i.e. set my$name)
  - No loops or conditionals

- **VTK classes (and member functions) must be registered or compilation fails**

- **Use WATT for a head start on the final C++ program.**
  - Unsupported content can be added by hand after rest is compiled.
  - Charge Density service followed this approach.
Templates

- C++ code fragments
- Variables
  - $$VARIABLE_NAME$$
  - replaced with buffered content from WATT
- Contain pre-written standard methods
- Structure and content of template can change; default template structure shown here

{Includes}
{WATT_CLASS Definition
 [Global Declarations]
 [Constructor]
 [Member Functions]}
{Main Method}
{Web Method Wrappers}

NOTE: each line can contain one or more variables
package require vtk
{
...
}
vtkQuadric quadric
  quadric SetCoefficients .5 1 .2 0 .1 0 0 .2 0 0
{
...
}
vtkContourFilter contours
  contours GenerateValues 5 0.0 1.2
{
...
}
#WATT_EOF
iren AddObserver UserEvent {wm deiconify .vtkInteract}
iren Initialize
wm withdraw .

NOTE: original is 33 lines (65 w/ comments);
WATT output is 300+ lines C++
Simple Example (Cont’d)

// Includes:
#include "vtkQuadric.h"
#include "vtkContourFilter.h"
(...)

// WATT_CLASS Definition:
class MyWattClass {

// Global Declarations:
vtkQuadric *quadric;
vtkContourFilter *contours;
(...)

// Constructor:
MyWattClass () {
(...)
quadric = vtkQuadric::New();
quadric->SetCoefficients((float) .5,
  (float) 1, (float) .2, (float) 0, (float)
  .1, (float) 0, (float) 0, (float) .2,
  (float) 0, (float) 0);
(...)
contours = vtkContourFilter::New();
contours->GenerateValues((int) 5, (float) 0.0,
  (float) 1.2);
(...)
}

// Member Functions:
vtkUnsignedCharArray* render () {
(...)
}

// Main Method:
int main( int argc, char *argv[]) {
(...)
  wattObject = new MyWattClass();
  runSerialSoap( port, "MyService.wsdl" );
}

// Web Method Wrappers:
int ns__render (SOAP* soap,
  ns__renderResponse* result) {
(...)
}
WATT Live

- Perl CGI scripts to test custom Tcl with WATT
- Generated service available as download.
- Port range for testing services (and clients)
- VisQuad service image shown here
Charge Density Service

- 400+ line Tcl script for VLab
  - View Plane-Wave self consistent field (PWscf) charge densities as volume, isosurface and orthoslice representations.
  - Reads Gaussian Cube format

- Procedures to toggle representations and update values

- Original test for WATT
  - Available as example for WATT Live
Charge Density Portlet

- Gridsphere portlet provides controls; Axis servlet in back-end calls service.
- Manually written
- Dashed lines distinguish UI components.
- Orthoslice depth control depends on manual changes to back-end service.
Automatic Client Generation

- `#WATT_GUI_{}{...}` directives within “proc” content describe UI components
  - SETVALUE
  - PRESET
  - TOGGLE
  - RENDERED_IMAGE
- Components common to most UI packages
  - HTML, Qt, Swing, etc.
- Ruby-on-Rails utility (work in progress) uses WATT generated file to create Ajaxified web interface
**Kill-A-WATT (KWATT)**

- Work in progress to replace WATT
- Stand-alone web service
  - Service spawns new services
- Services (C++) control Tcl interpreter
  - Tcl was written to extend C++
  - Interpreter provides full access to Tcl types and commands without limitation
    - No registers or type hints
    - Use any package with Tcl bindings
      - Automate compute service generation
- Procedure content can be registered in the interpreter at any time
  - Patch/update service methods without downtime
Recap

- **WATT**
  - Automates service development.
  - Many limitations still present. KWATT to address these problems.

- **WATT Live**
  - Compiler as a service

- **Client Interface Generation**
  - Create user interfaces based on stub description
  - Possible to generate local and web based interfaces from one description
Acknowledgements

- Thanks to the Virtual Laboratory for Earth and Planetary Materials (VLab) for their support
  [vlab.msi.umn.edu](http://vlab.msi.umn.edu)
- This work is supported by NSF through the ITR grant NSF-0426867