WATT: A Compiler for Automated Visualization Service Generation

Evan F. Bollig
School of Computational Science
Florida State University
Tallahasse, FL 32306
+1 850 644 7018
bollig@scs.fsu.edu

Martin D. Lyness
Minnesota Supercomputing Institute
University of Minnesota
Minneapolis, MN 55455
+1 612 624 6730
linux@msi.umn.edu

Gordon Erlebacher
School of Computational Science
Florida State University
Tallahassee, FL 32306
+1 850 644 0186
erlebach@scs.fsu.edu

David A. Yuen
Minnesota Supercomputing Institute
University of Minnesota
Minneapolis, MN 55455
+1 612 624 9801
davey@msi.umn.edu

ABSTRACT

Service-oriented application development is a time consuming task that changes little between projects. In general, development is a two step process where developers first create the core functionality of the service and then routinely integrate details for distribution, transport protocols and user interfaces. Addressing a need to port standalone applications to web services, we have developed a compiler to automate the generation of Web Services. This compiler is called the Web Automation and Translation Toolkit, or WATT. Originally designed as a utility for porting Tcl scripts for the Visualization Toolkit (VTK) to equivalent but more efficient C++, WATT has been extended to seamlessly integrate the SOAP transport protocol to create working visualization web services. Within this document, we present details of the WATT compiler, including motivation, goals and example applications.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *Code generation,* I.3.8 [**Computer Graphics**]: Applications

General Terms

Algorithms, Performance, Experimentation, Human Factors

Keywords

Web Service, Automation, SOAP, GUI, Visualization.

1. INTRODUCTION

Service-oriented architectures that address workflow execution and problem solving are becoming increasingly popular for scientific computing. As seen in the architectures of the VLab [26], TeraGrid [31] and eMinerals [8] initiatives, many issues related to Grid computing and solving scientific problems have already been investigated. While such consortiums investigate specialized topics and approach solutions differently, their general goals have considerable overlap, such as task automation, public access to utilities and distributed computation that conform to standards. Task automation pertains to the development of workflows, load balancers, compilers and other technologies that take control from users and abstract the complexities of the underlying work. Public access to utilities usually refers to public repositories from which they can be downloaded, or web interfaces to control the execution of such utilities from a remote location.

Working with the VLab consortium, we have concentrated on the development of visualization tools, including 2D and 3D representations through hand-crafted, install-on-demand, Java WebStart [18] clients; and simple web page interfaces that utilize remote web services to perform post-processing and data rendering. In the case of WebStart clients, data is stored in remote repositories, downloaded to the client and visualized with packages such as VisAD [14], Prefuse [13] and the Java bindings for OpenGL (JSR-231/JOGL) [16]. To better leverage existing desk-side approaches to visualization, we have created web services that translate scripts written to drive the free and popular Visualization Toolkit (VTK) [28] libraries. These services implement a client-server paradigm where one or several clients connect to a server, which handles most of the computational load.

The development of VTK services generally follows a simple process. First, a VTK Tcl script is written to render data correctly. Second, the Tcl is translated by hand into equivalent but more efficient C++ using a one-to-one mapping between methods in Tcl and methods in C++. Third, additional code is integrated to enable communication via the SOAP protocol and to define web-methods. Finally, resulting source is compiled into a complete web service executable (on the server side).

Since this process does not change from one service to the next, we automated the above four steps with a newly developed compiler referred to as WATT (the Web Automation and Translation Toolkit). With WATT, developers need only concern themselves with client side programming with no knowledge of client-server communication. All features of the WATT compiler help the developers concentrate solely on their data representations; WATT takes care of the rest.

Ideally, developers of distributed applications should develop code on a local machine and leave the details of distribution, communication and integration. At this stage, we do not claim to have a full solution to the problem of automatic software generation; however, the WATT compiler already handles many of the issues related to automating the generation of web services from standalone input code. WATT translates standalone VTK based Tcl scripts into C++ and automatically integrates the

Bollig, E. F., Lyness, M. D., Erlebacher, G., Yuen, D. A., WATT: A Compiler for Automated Service Generation, International Workshop on Grid Computing Environments 2007, Nov., 11-12, 2007, Reno, NV, USA. An electronic version of this document will be available at http://casci.rit.edu/proceedings/gce2007

necessary code to form a complete visualization web service. In this paper, we introduce the WATT compiler, describe its features, design and implementation, and illustrate WATT in action through examples of the compiler's use within VLab. We also present an ongoing effort to automate the generation of graphical client interfaces based on directives embedded in the input Tcl code.

2. RELATED WORK

The concept of the WATT compiler is not unique. For example, the language-to-language compiler SWIG [4] has the ability to compile standard C/C++ code and produce wrapped code for a variety of output languages including Tcl, Python, Ruby, etc. However, SWIG does not support the compilation of Tcl to C++, a common problem with all compilers found in our effort to use existing software.

Since the Tcl language was written in C++, it seems logical to use the low level Tcl libraries and headers directly, or a higher level abstraction like C++/Tcl [9], MkTclApp [25] or Bridge-Tcl [5] that allow C++ to call Tcl and vice-versa. However, in our push for efficient visualization web services we do whatever it takes to cut corners and optimize code. Using Tcl from within C++ requires evaluation of the scripted language within the Tcl interpreter; translating Tcl to C++ bypasses the interpreter and executes directly. Translating to C++ directly also forces us to choose C++ SOAP libraries over Tcl implementations like TclSOAP [30], tWSDL/TWiST [33] or Gerald Lester's Web Services for Tcl [22].

Related to the automation of service deployment, the Enunciate [11] project lets developers write their web service code in Java, and then completes the task of deployment, servlet configuration, packaging and interoperability with other services/clients. The Enunciate project automatically generates SOAP, REST and JSON endpoints for every service, but at the time of this writing still requires users to define and complete the web methods themselves. WATT uses the input script to generate the web methods for SOAP only, and depends on WATT Live to deploy the service.

3. TECHNOLOGIES USED

The Visual Toolkit (VTK) is a visualization package that is free, quite popular and feature-rich. It supports a wide range of visualization techniques, such as volume rendering, isosurfaces and streamlines, many of which take advantage of the hardware of latest commodity video cards. Both on-screen and off-screen rendering are supported. VTK is well supported on most operating systems (Windows, Linux, OSX, etc.) In addition to the above, VTK includes a wide array of input and output modules to handle the most commonly used data formats for bitmaps (JPEG, PNG, etc.) or 3D models (VRML, CAD, etc.). In addition, the C++ VTK sources can be optionally compiled to include bindings for Tcl, Python, and Java. Other projects provide interfaces to VTK from Ruby [36], and Perl [12]. The developers of VTK have expended substantial effort to make their software usable by a wide audience.

The Tool Command Language (Tcl) [32] is a high level scripting language quite popular for interfaces to scientific and visualization toolkits like VTK, Amira [2] and VMD [15]. Tcl is loosely typed and seldom differentiates between number values, logic, strings or objects. In addition, everything in Tcl is considered a command; commands instruct invocation of procedures on values, evaluation of logic, instantiation/construction of objects, etc. The structure of Tcl makes it easy to expand the available commands by writing procedures in other languages. Since Tcl is less concerned with type than C++, writing simple applications is quite straightforward. There is no need to include headers, manage memory, or declare and coerce types for all functions and calls. Also, since it is a scripting language, modifications to Tcl programs provide developers instant feedback with either a working product or prototype code.

SOAP-enabled C++ services are implemented with the gSOAP library [34]. gSOAP is a highly efficient toolkit designed for SOAP message passing in codes that require high performance. gSOAP requires some input from the user, specifically, a description header to specify web-method signatures, types and serializable objects (i.e., objects that can be converted to XML). In addition, users also provide the content of each web-method, but not the code that binds a SOAP request or response with the method. A compiler ("soapcpp2") included in the gSOAP distribution, consumes and compiles the provided header and creates wrappers to connect incoming or outgoing messages with the specified serializable types and available web-methods. gSOAP outputs separate C++ headers and implementations that integrate seamlessly with either a service (callee) or a client (caller). This generated code also includes details of the low level socket connections that enable two-way messaging. Most aspects of SOAP and low level communication are hidden from the users. Unfortunately, users must still create their own serializable types, and specify to SOAP when to listen for an incoming message. gSOAP interprets what is received and either distributes it to the correct RPC function or the user can intervene and handle the message. With automatically generated client- and server-side bindings for C++ programs, gSOAP is an important component in the complete automation of the transport layer. WATT aims to go the last mile and remove the user from all aspects of web service generation, *except* writing the initial visualization Tcl script.

4. WATT ARCHITECTURE

Figure 1 illustrates how WATT transforms a Tcl script into a fully functional web service. There are four steps. First, the input script is submitted to the WATT compiler. WATT converts the script into two types of output files: the first contains a C++ class with methods that are direct translations of the Tcl procedures. In addition, WATT generates stubs that wrap the aforementioned methods to be called by SOAP. These stubs are referred to as web-methods. The second type is a collection of gSOAP sources required by the gSOAP compiler to build the web service. The gSOAP sources include the web-methods description header,

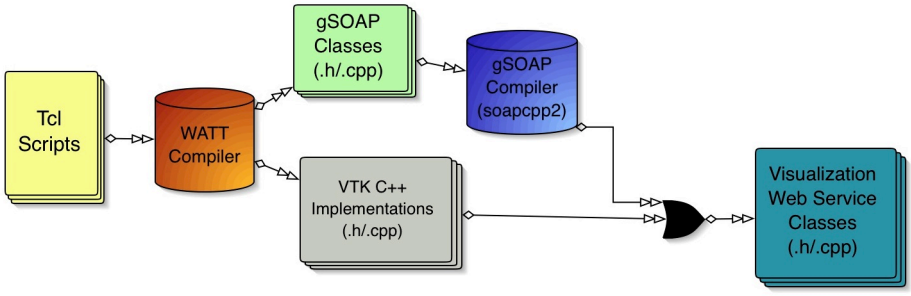

Figure 1: WATT execution workflow

other files containing the declaration and implementations of standard serializable objects (including Vector, Matrix and Base64Binary types) and the WATT header, which provides methods to automatically set up threads to handle SOAP messages. The web-method description header is then sent to the gSOAP compiler, which generates the necessary sources to generate, parse, send and receive SOAP messages. Finally, all generated sources from WATT and gSOAP are consolidated. Together they form a working web service implementation that can be compiled by most C++ compilers.

WATT begins Tcl conversion by sending the input code through both a custom-written parser and a type-checker. Based on immediate contextual use, WATT can determine the type for most of the tokens within the script. Type and context information are stored in an abstract syntax tree generated during a first pass through the script. Once the entire file is parsed, WATT rescans the file one or more times using the latest tree information until either all types are determined or the undetermined types are tagged as "WATT_AMBIGUOUS", which in turn generates an error on compilation. In this case, the user can choose to manually update the source. A typical case where inferring type is not possible are function return values.

When a service is generated from the Tcl input script, WATT creates a list of stubs that wrap all Tcl procedures ("proc" commands) as gSOAP web methods. The resulting WSDL contains references to the wrapping stubs and also to the default methods defined for all services generated by WATT. Any dependencies between procedures (i.e., one procedure may depend on another to be called first) are not represented in the WSDL. Instead, dependencies are handled by the author of the Tcl script by either calling them in the first few lines of the dependent procedure contents, or by calling the dependencies outside of the procedure declarations. Tcl interprets any command outside of procedures or functions as global commands that are evaluated once. When WATT parses such commands it places them in an initialization function in the service and they are executed once upon startup. In addition, our visualization services are not stateless; they maintain state to avoid unnecessary loading of data files, rendering, and other time-consuming tasks. Developers should use this to their advantage when writing scripts.

Given that WATT does not support all the features of Tcl, we avoid a direct translation from Tcl to binary code. Instead, the Tcl script is translated to C++. This gives users the opportunity to enhance the resulting C++ with more general constructs that could not be translated, due to an incomplete Tcl to C++ implementation. Missing support includes loops, conditionals, variables embedded within strings (e.g. "hello $name"), and commands or names containing variables (e.g. set my$name value). The C++ source and automatic build instructions make it easy for users to generate complex web services by first creating a simplified Tcl script with method names and simplified implementations, automatically translating these scripts to C++ and associated SOAP routines, and then changing the C++ method implementations according to desired specifications. The generation of the charge density service created for VLab follows such an approach.

Once the Tcl has been typed, WATT uses a registry and interrupt system to generate the C++ source code. The registry is a collection of associations between expected formats and commands in Tcl and C++. The interrupt system allows the default translation of methods within WATT to be overridden. For example, regular C++ constructors are invoked with the "new className()" format, whereas standard VTK constructors are called using "vtkName::New()". An interrupt catches calls to VTK constructors and translates them to the C++ format.

Another use of the interrupt system allows for the generation of web method wrappers and a gSOAP description header based on specific Tcl input patterns, rather than generate one web method wrapper for each Tcl procedure, which is the default behavior. Recall that a web method is callable from the final web service. For example, one might only publish methods that contain a certain prefix, suffix or keyword. The gSOAP bindings are entirely optional, which allows users to first develop an interactive Tcl script, and let WATT compile it into an interactive C++ application. When developers are sure the output C++ application works correctly, the compiler can merge in the bindings for gSOAP. It is important to note that WATT also provides a uniform set of methods that are available in all generated services through the use of a templating system.

The format of the output files generated by WATT is not hard-coded, but is instead read in from a template file at run time. The template is a C++ code fragment that contains directive variables to specify the locality of content. Variable names appear as "$$VARIABLE_NAME$$" within the template. When WATT translates Tcl to C++, it stores the generated content in various output buffers, each buffer corresponding to a specific variable name. WATT then reads the template into memory and replaces all variables found in the template with the corresponding buffer contents. Typical templates control where WATT places function declarations, header files, global variable declarations, and the body of the code. Different templates are used if WATT is compiling a web service or a standalone application. The web

service template defines the set of standard web-methods for all WATT generated services.

CMake [23] manages all generated source files generated by WATT and gSOAP, in addition to various operating system dependencies. Once an inventory of available resources is complete, CMake generates a Makefile to build binaries as suggested in a description file. A Tcl script designed to be a web service could easily have 20 or more web-accessible procedures. Prior to compilation by the gSOAP compiler, these procedures are converted to C++ within one file and joined by at least five supplementary files including gSOAP and user interface descriptor files. With 20 or more procedures accessible from the web service, the gSOAP compiler can easily generate 40 additional source files.

The CMake file requires all binaries to depend on the shared library, libwatt.so, provided with WATT. This library contains definitions for a core set of VTK classes, serializable objects, etc. for general use. For example, "CurlUtility", which is an abstraction to the cURL library [10] to allow easy fetching of remote files to the service's host. We discuss an example VTK class included in the library in the next section.

Finally, the generation of visualization services using WATT can be accomplished using WATT Live [37], which is a set of Perl based CGI scripts that allows users to upload Tcl scripts typed into the input form in a browser. WATT is executed on the input data and users receive feedback about the compilation process. All output files generated by WATT can be downloaded so that users can compile and install the generated service on their own web servers. Alternatively, WATT Live can compile and execute the service on our server. The latter option is primarily intended for verification of functionality and debugging rather than for production deployment and use. All services generated by WATT have an internal counter to shut the service down after one hour of inactivity.

When a service is deployed by WATT Live, users can also test a client connection to the service as shown in Figure 2. Perl scripts control the execution of a C++ client program output by WATT and compiled against the gSOAP generated bindings. The client is only aware of the default web-methods common to all WATT generated services, and the WATT Live page calls only two of them: one to monitor the service's visualization, and the second to change the camera position. The Tcl code used to create the service is available from the WATT Live front page as an example of supported Tcl syntax, and to demonstrate the output generated by the WATT compilation process.

Security was not specifically addressed during the development of WATT. Instead, the assumption was that users of WATT would be responsible for integration into their own secure environment. The initial emphasis was to deliver a functional product, followed by support for additional types and syntax. If desired, users have the option to modify WATT's templates to enhance security, or they can modify generated services individually; however, out of the box, Web Services generated by WATT utilize only the default mechanisms provided by gSOAP. Security can also be achieved by integrating WATT within an existing secure portal. In our case, the intent is to include WATT as a feature in the VLab portal. It should be noted that WATT does not currently support the translation of potentially dangerous commands such as "exec" or "interp". For syntax supported by WATT, WATT Live allows the generated service to be compiled and tested on a small range of ports on our test server.

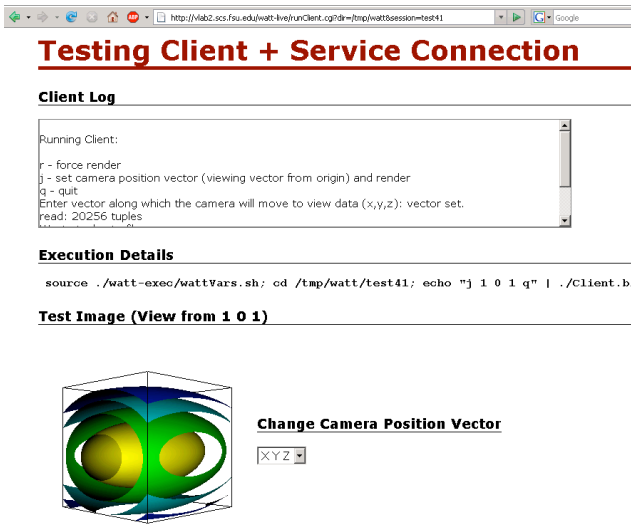

Figure 2: Testing the WATT compiled VisQuad.tcl with WATT Live

5. EXAMPLE APPLICATIONS

Two example applications are now considered to demonstrate functionality and discuss restrictions placed on scripts by WATT. First, a simple example helps readers understand the input and output structure. Second, an application of WATT within the VLab consortium is considered. For more details readers are referred to the WATT Live webpage [37] where the Tcl examples are available to study in detail and can be compiled into C++ source.

5.1 VisQuad

WATT was written as a VTK Tcl compiler/translator; therefore, it is best to start with an example application taken from the VTK documentation. To this end we comment on the compilation of VisQuad.tcl [28], a standard example in VTK distributions. The script is 33 lines of executable Tcl and demonstrates contour filtering applied to an implicit function for 3D isosurface representations. Figure 2 shows the results returned by the VisQuad service using WATT Live.

Compiling VisQuad.tcl with WATT requires the "#WATT_EOF" directive to be added before the last three lines of the script. This directive indicates to WATT that it should assume the end of file has been reached and it no longer needs to parse Tcl. It is interpreted by Tcl as a comment and has no effect on normal script execution. Standalone Tcl scripts can be written as a combination of Tcl and Tk commands where Tcl manages computation details and Tk manages user interfaces. WATT does not support Tk commands. By integrating the directive into a script, Tk can generate a user interface in standalone mode, and WATT can finish parsing the file for a web service without errors.

When WATT compiles a script, the output C++ is organized in the format described in Listing 1. For every VTK type that appears in the script, a corresponding header entry is placed in the "Includes" section. WATT embeds the translated Tcl within a C++ class type. Commands not contained in a Tcl "proc" are

called from the C++ class "Constructor" to imitate Tcl's single execution of the same commands. Any variables or objects used in the constructor become a part of "Global Declarations" and are accessible to "Member Functions" (direct translations of Tcl procedures). The "Main Method" initializes an instance of the WATT class and listens for SOAP messages. When messages are received, they are distributed to a web-method in the "Web Method Wrappers" section where a wrapping method calls its corresponding member function in the WATT class, converting output to a serializable type known by gSOAP.

Listing 1: General Structure of WATT Output

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

Listing 2 provides output snippets from the application of WATT to VisQuad.tcl. Comments in bold face show code structure. Lines 8 and 10 demonstrate the use of type casting to match function input parameters to C++ type signatures. VisQuad.tcl does not contain procedures; thus, Member Function and Web Method Wrapper sections contain only the set of web-methods common to all WATT services. The web-method on line 20, "ns__render", calls the method "render" on line 12. Inside the web-method, the returned "vtkUnsignedCharArray" type is converted to a byte array and sent to gSOAP under the alias "ns__renderResponse". The function on line 17 executes the gSOAP initialization and starts an infinite loop for requests on the specified port.

5.2 Charge Density Visualization

The primary motivation behind developing WATT is to create visualization services for the VLab consortium. One such example was a 400+ line Tcl script that provides Volumetric, Isosurface, and Ortho-sliced renderings for scalar values of charge density distributions computed by PWscf [27], the core computational engine of VLab. Although various programs like XCrysDen [21] and JMol [20] are quite capable of rendering the output file in various modalities (i.e., crystal structure or atom positions), none provide the customizable versatility of VTK nor the ability to render visualizations off-screen as a web service. One of the requirements of VLab is to provide all web services within a web portal. In response to this, we applied the WATT compiler to the Tcl script and produced a service specialized for charge density visualization.

The service accepts standard Gaussian Cube [7] input that describes primitive cell vectors, atom positions and uniformly spaced volumetric data corresponding to charge within the cell. At the time of this writing, VTK version 5.0 has a standard Gaussian Cube reader built-in, which we found to improperly transform the

Listing 2: Partial C++ Output Generated for VisQuad.tcl

```
// Includes
1.   #include "vtkQuadric.h"
2.   #include "vtkContourFilter.h"
       (…)
// WATT_CLASS Definition:
3.   class MyWattClass {

// Global Declarations
4.       vtkQuadric *quadric;
5.       vtkContourFilter *contours;
           (…)
// Constructor
6.       MyWattClass () {
           (…)
7.         quadric = vtkQuadric::New();
8.         quadric->SetCoefficients((float) .5, (float) 1, (float) .2,
     (float) 0, (float) .1, (float) 0, (float) 0, (float) .2, (float) 0,
     (float) 0);
             (…)
9.         contours = vtkContourFilter::New();
10.        contours->GenerateValues((int) 5, (float) 0.0, (float)
     1.2);
             (…)
11.    }                  // end Constructor

// Member Functions:
12.      vtkUnsignedCharArray* render () {…}
           (…)
13.  } *wattObject;

// Main Method:
14.  int main( int argc, char *argv[] )
15.  {
           (…)
16.      wattObject = new MyWattClass();
17.      runSerialSoap( port, "MyService.wsdl" );
18.  }

// Web Method Wrappers:
19.  int ns__render (SOAP* soap, ns__renderResponse*
     result) {…}
       (…)
```

input data. Unfortunately, the default reader ignores the coordinate system specified by primitive vectors within the file and assumes the data to be in Cartesian coordinates. We substituted a home grown reader. The class is available through the core library, "libwatt.so", included by all programs generated by WATT. Since the routines we use from VTK depend on the Cartesian coordinate system, our reader linearly interpolates data from its original oblique coordinate system into a Cartesian representation. Data are shown in a orthorhombic box that assumes that the primitive cell (based on the Face Centered Cubic structure [29] is periodic in all directions and then clipped to the dimensions of the bounding box. Figure 3 shows a simple Silicon diamond structure created with high-energy cutoffs on a fine mesh of 729,000 points and illustrates the diamond-shaped primitive cell (gray tube wire frame). Contours are drawn in the clipped cube box (thin white bounding box). Figure 4 shows an example

of orthorhombic structure in Perovskite data ($MgSi0_3$), whose primitive cell is identical to the clipped box. Figures 3 and 4 were produced by the on-screen interactive C++ code translated from Tcl by WATT. Output produced off-screen by the completed web service is the same but requires a client application/servlet to display the images.

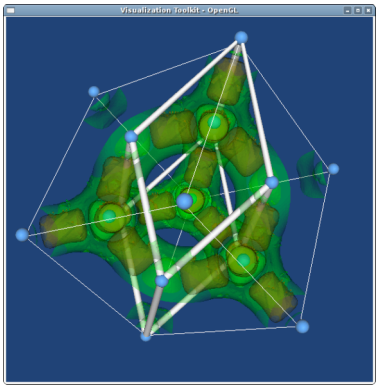

Figure 3: Isosurface representation of simple Silicon diamond rendered by the charge density application output by WATT.

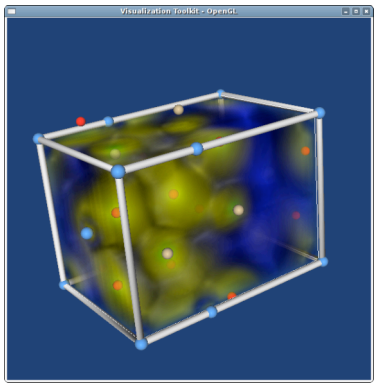

Figure 4: Volumetric representation of Perovskite ($MgSiO_3$) primitive cell as produced by charge density application output by WATT.

Most Web services require a graphical user interface before they become widely adopted. We have designed a simple web-based user interface to the charge density application, shown in Figure 5. The implementation only depends on a JavaScript-enabled web browser. Java and JSP [17] were used to implement the interface as a portlet within the VLab portal. Simulations can be run from within the portal and the results immediately rendered within the same window. The different modules—indicated by a dashed border—connect to different web methods. Functionality is limited at this time; however, the available options include loading of new data files, on/off toggling of options, data slicing with a cut plane with translation of the slice, and selecting the viewing direction. Changes to any of the modules result in a JavaScript call to a backing Java servlet that in turn makes a call to the web service using the Apache Axis libraries [3].

The modular design of the handwritten Java portlet comes from our current efforts to identify reusable interface components and automate not just service-, but also client-generation with WATT. For example, html input forms composed of one or many input text fields plus a submit button can be used to interface with methods that require one or many input parameters. Toggles (radio buttons or checkboxes) correspond to a web-method with a single boolean input, or two web-methods that turn representations on or off respectively. Using directives embedded within Tcl comments, we have been investigating the auto-generation of a Ruby-on-Rails interface, shown in Figure 6. Client interface directives are specified using comment lines with the structure "#WATT_GUI_{...}". Current suffixes recognized by WATT include: ``SETVALUE'' for a form (text fields and submit button) that allows user input, ``PRESET'' for predefined values in a pull-down menu (does not allow user input), "TOGGLE_ON" and "TOGGLE_OFF" for toggle switches (radio buttons or checkboxes), and "RENDERED_IMAGE" for displaying an image. To bind a module with a web-method, the directive must occur within the Tcl procedure statement. As WATT parses the input Tcl script, it filters off directives and writes them to a separate file paired with the associated C++ web-method declarations. This file is read by a Ruby script that parses off the directive information and associated web-method name, parameters and return type. Ruby-on-Rails makes it easy to generate Ajaxified HTML interfaces that provide improved interactivity over standard web interfaces.

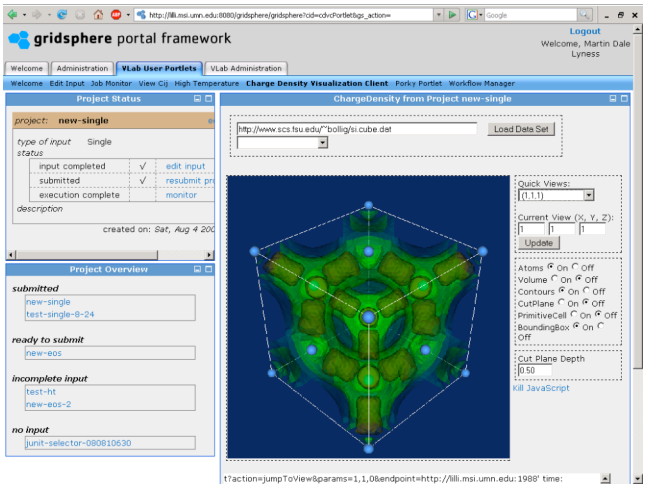

Figure 5: Portlet interface to the charge density service created for the VLab collaborative computational portal.

6. FUTURE WORK

Interactive remote visualization through web services has proved rather challenging. In our current implementation of WATT, output services depend on clients to invoke a render method and download resulting bitmap images. This view-on-demand method allows clients and services to be less synchronous, but the frame rate is rather low (< 3 fps), inefficient and fails to provide fully interactive manipulation of objects. In addition, recall that WATT generated web services maintain state. Research into AJAX and portal based client interfaces have motivated this setup, but it is clear that improvement is necessary. Our future plan is to experiment with highly synchronous clients and services that initiate communication via an asynchronous web service protocol like SOAP, and open direct socket connections to allow a client to receive streaming images from the service similar to a VNC connection. A second connection would pipe client input (mouse and keyboard) to the server for interactivity.

We have initiated a project to investigate the auto-generation of client graphical interfaces. We are starting with Ruby-on-Rails due to its strong support for databases and Ajax. Once the GUI is automatically generated, it will be natural to develop interface skins and store these in databases for use by the user community. We are also looking into automatic unit testing of the auto-generated service.

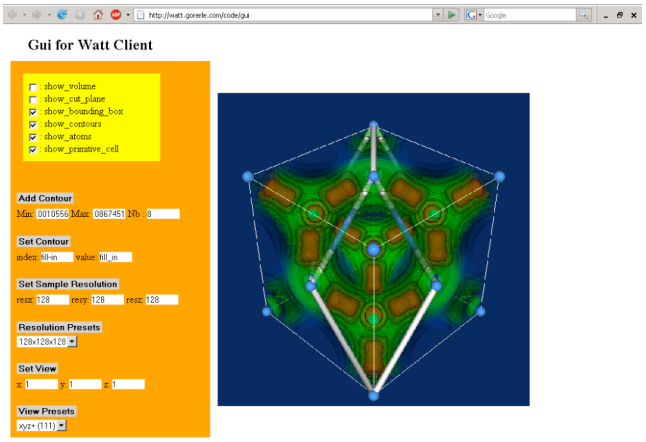

Figure 6: Initial results of Ruby-on-Rails interface.

Also in development is Kill-A-Watt (KWATT), to alleviate some of the deficiencies of WATT. KWATT is an attempt to avoid details of cross-language compilation by using the official Tcl interpreter via the C++/Tcl library. The C++/Tcl Library allows bi-directional communication between a C++ application and the Tcl environment. KWATT is not a compiler like WATT, but a stand-alone web service with a set of web-methods published via gSOAP. Part of KWATT's desired functionality is the ability to spawn a new web service to evaluate input Tcl. Currently, during evaluation of a Tcl script KWATT interprets the code structure and integrates communication details directly via evaluated Tcl calls to the TclSOAP library. Since KWATT controls the Tcl interpreter, it has access to the full list of Tcl commands and types and can load additional commands from other packages like VTK. When KWATT consumes Tcl input to generate web methods, the Tcl interpreter extends the list of commands previously defined. This implies that KWATT has the potential to be patched or updated remotely without downtime.

Preliminary work on Kill-A-WATT has demonstrated it is more promising for development to evaluate scripts directly within the Tcl interpreter controlled by a C++ application than to continue translating from Tcl to C++. The design of KWATT leverages commands, types and packages already available to the Tcl interpreter. Support for the SOAP protocol is limited in Tcl—TclSOAP is more intended as an RPC package than for message passing, even lacking a published WSDL. In light of this, KWATT connects service methods to the gSOAP library and when invoked, these C++ methods evaluate corresponding Tcl procedures within the Tcl interpreter. To broaden application areas, we are investigating the use of other C++ libraries including Boost.Python [1] to interpret additional scripting languages. One goal of KWATT is to allow developers to choose the most appropriate language for their core code and then automatically configure the corresponding interpreter and SOAP bindings for a complete web service.

7. CONCLUSION

In summary, the WATT compiler is a utility for automatically generating visualization web services based on standalone VTK Tcl scripts. We have explained our motivation to originally develop WATT as a Tcl to C++ translation utility, and later as a tool to automatically integrate the SOAP protocol into translated code. Design and implementation details were discussed including a brief presentation of WATT's web-based front-end, WATT Live, and work in progress to automatically generate client user interfaces based on directives within the Tcl input. We also presented an application of WATT to generate a visualization web service for charge density output by quantum calculations within the VLab consortium's web portal.

8. ACKNOWLEDGEMENTS

Many thanks to Pierre Carrier for the charge density data and info. Thanks as well to the rest of the VLab group for their support and feedback on this project.

This work is supported by NSF through the ITR grant NSF-0426867.

9. REFERENCES

[1] Abrahams, D., Grosse-Kunstleve, R. W., "Building Hybrid Systems With Boost Python", Boost Consulting, 2003

[2] Amira Homepage: 3D Data Visualization, http://www.amiravis.com

[3] Apache Axis, http://ws.apache.org/axis

[4] Beazley, D. M. 1996. SWIG: an easy to use tool for integrating scripting languages with C and C++. In Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4 (Monterey, California, July 10 - 13, 1996). USENIX Association, Berkeley, CA, 15-15.

[5] Bridge-Tcl. http://sourceware.org/sid/component-docs/bridge-tcl.html

[6] Bollig, E.F., Jensen, P.A., Lyness, M.D., Nacar, M.A., da Silveira, P.R.C., Kigelman, D., Erlebacher, G., Pierce, M., Yuen, D.A., da Silva, C.R.S. 2007, VLAB: Web Services, Portlets, and Workflows for Enabling Cyber-infrastructure in Computational Mineral Physics, Physics of the Earth and Planetary Interiors (2007), DOI=http://dx.doi.org/10.1016/j.pepi.2007.03.005

[7] Bourke, P., Gaussian Cube Files (Dec. 2003), http://local.wasp.uwa.edu.au/~pbourke/dataformats/cube

[8] Calleja, M., Bruin, R., Tucker, M. G., Dove, M. T., Tyer, R., Blanshard, L., van Dam, K. Kleese, Allan, R. J., Chapman, C., Emmerich, W., Wilson, P., Brodholt, J., Thandavan, A. and Alexandrov, V. N. 2005, Collaborative grid infrastructure for molecular simulations: the eMinerals minigrid as a prototype integrated compute and data grid. Mol. Simulat. 31 (2005), p. 303.

[9] C++/Tcl, http://cpptcl.sourceforge.net

[10] cURL and libcurl, http://curl.haxx.se

[11] Enunciate, http://enunciate.codehaus.org

[12] Graphics::VTK - Perl Interface to Visualization Toolkit, http://search.cpan.org/~cerney/Graphics-VTK-4.0.001/VTK.pm

[13] Heer, J., Card, S. K., and Landay, J. A., Prefuse: A Toolkit for Interactive Information Visualization, ACM Human Factors in Computing Systems (CHI), 421-430, 2005

[14] Hibbard, W., Rueden, C., Emmerson, S., Rink, T., Glowacki, D., Whittaker, T., Murray, D., Fulker, D. and Anderson, J., Java distributed components for numerical visualization in VisAD. Communications of the ACM, Volume 48, Issue 3, 2005, pp.98-104. DOI=http://doi.acm.org/10.1145/1047671.1047676

[15] Humphrey, W., Dalke, A. and Schulten, K., VMD - Visual Molecular Dynamics, J. Mol. Graphics, 1996, vol. 14, pp. 33-38. http://www.ks.uiuc.edu/Research/vmd/

[16] Java Bindings for OpenGL (JOGL), https://jogl.dev.java.net/

[17] Java Server Pages Technology, http://java.sun.com/products/jsp/

[18] Java WebStart Technology, http://java.sun.com/products/javawebstart/

[19] Jensen, P A, Bollig, E F, Yuen, D A, Erlebacher, G, Momsen A R (2006). Automatic Generation of Remote Visualization Tools with WATT, EOS Trans. AGU, 87(52), Fall Meet. Suppl., Abstract IN14A-06 2006.

[20] JMol: an open-source Java viewer for chemical structures in 3D, http://jmol.sourceforge.net/

[21] Kokalj, A., Computer graphics and graphical user interfaces as tools in simulations of matter at the atomic scale, In Proceedings of the Symposium on Software Development for Process and Materials Design, Comp. Mater. Sci. 28(2), 155-168 (Oct. 2003). DOI=http://dx.doi.org/10.1016/S0927-0256(03)00104-6

[22] Lester, G., Web Services for Tcl, http://members.cox.net/~gerald.lester/WebServicesForTcl.html

[23] Martin, K., Hoffman, B., Mastering CMake 2.2, Kitware, Clifton Park, New York. 2006. ISBN 1-930934-16-5.

[24] Mesa 3D Graphis Library, http://www.mesa3d.org/

[25] MkTclApp: A Tool for Mixing C/C++ with Tcl/Tk, http://www.hwaci.com/sw/mktclapp/

[26] Nacar, M., Aktas, M., Pierce, M., Lu, Z., Erlebacher, G., Kigelman, D., Bollig, E. F., De Silva, C., Sowell, B., and Yuen, D. A., VLab: Collaborative Grid Services and Portals to Support Computational Material Science, Concurrency and Computation: Practice and Experience 19(12), 1717-1728 (2007). DOI=http://dx.doi.org/10.1002/cpe.1199

[27] PWscf (Plane-Wave Self-Consistent Field), http://www.pwscf.org/

[28] Schroeder, W., Martin, K. and Lorensen, B., The Visualization Toolkit, An Object-Oriented Approach to 3D Graphics (4th edition), Kitware (2002). ISBN 1-930934-19-X

[29] Sque, S., Structure of Diamond (Mar. 2006), http://newton.ex.ac.uk/research/qsystems/people/sque/diamond/structure

[30] TclSOAP, http://tclsoap.sourceforge.net/

[31] TeraGrid, http://www.teragrid.org/

[32] The Tool Command Language (Tcl), http://www.tcl.tk/

[33] tWSDL, http://code.google.com/p/twsdl/

[34] van Engelen, R. A. and Gallivan, K., The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002), 128-135, (May 2002), Berlin, Germany.

[35] VLab Homepage, http://vlab.msi.umn.edu

[36] VTK-Ruby, http://www.gfd-dennou.org/arch/ruby/products/ruby-vtk/

[37] WATT Live, http://vlab2.scs.fsu.edu/watt-live, http://lilli.msi.umn.edu/watt-live

[38] WATT Homepage, http://vlab.sourceforge.net/WATT