Mulini : Code Generation tool for Automated Staging Application System

Deepal Jayasinghe
College of Computing, Georgia Institute of Technology
deepal@gatech.edu
GT Number : 902531132

ABSTRACT

The increasing complexity of enterprise and distributed systems demands automated design, testing, deployment, and monitoring of applications. Testing, or staging, in particular poses unique challenges. The idea behind Elba is to create automated staging and testing of complex enterprise systems before deployment to production. Automating the staging process lowers the cost of testing applications. Feedback from staging, especially when coupled with appropriate resource costs, can be used to ensure correct functionality and provisioning for the application. The key component of Elba is its code generator called Mulini, it extracts test parameters from production specifications and generate deployment artifacts and creates staging plans for the application. Depending on the feedback we get we can change the input parameters and regenerate the code and run the test.

Keywords

Elba, Staging, Code generation

1. INTRODUCTION

Increasing complexity of large enterprise and distributed application system make the management of the system is an increasingly important and increasing expensive technical challenge. Staging comes as a handy tool when solving those important challenges. Staging is a natural choice for large enterprise distributed system like data center environments where sufficient resources are available for adequate evaluation of the system. The key advantage is Staging provides system developers and system administrators to monitor and tune new deployment configuration under simulated work condition before the system goes into production.

The process of designing, development, staging, deployment, and production of complex enterprise distributed systems is complex task in itself. Especially staging engenders unique challenges, due to, first because of its role linking development and deployment activities and second because of the need for staging to validate both functional and extra-functional requirement of the system. This process is further complicated by multiple iteration, each much shorter in duration than expected for the application in production runtime.

In the traditional approach of staging is to usually a manual process, complex and time consuming fashion. When the complexity of the system increases the limitation inherent to manual approaches tend to decrease the possibility of effectively staging that same complex application. Moreover adoption of Service Level Agreement (SLA), Test Based Languages(TBL) and Workflows those define the requirement and performance of the system also complicate the staging system. When the staging process become complex then the limitation of the manual process make a major obstacle. As a result of that automating the staging process is important and challenging research area. Elba alone with Mulini code generator act as a automated tool for complex enterprise distributed systems.

In the previous researches of Elba project has found two important problems that the code generator need to be addressed. First encapsulating middleware for distributed information flow system, which are characterized by continuous volumes of information traversing a directed workflow network. Second the resource deployment problem, where distributed application should start efficiently and in provable correct order by simultaneously enforcing serialization constrains and leveraging the distributed system's inherent parallelism. Solution to both the problems was to use an approached called Clearwater [1], which maps evolving SLA, TBL and Workflow to multiple execution platforms.

The remainder of this paper is structured as follows. Section2 describes about application background, approaches and application technologies.In Section3 I have described the staging process in details, which included challenges, advantages and steps involved in the process. In section4 details the overview of Elba project and Section5 details Mulini code generator and implementation and evaluation of RUBBoS benchmark. We explain the links of our work to related research approaches in Section6 and conclude the paper in Section7.

2. APPLICATION BACKGROUND

This sections discuss background information of Mulini code generator. Mulini code generator follows so called Clearwa-

CS7001 2008 Main project

ter approach, and Clearwater approach is XSLT [2] transformation process with multiple intermediate steps. As a results of that Mulini is a code generation system built on XML[3] and XSLT, much more flexible and extensible. First part of this section discuss about architecture and internal process of Clearwater approach, next it discuss extensible nature of XML in the context of code generation and finally extensible and flexible nature of XSLT and the advantage of using XSLT in in code generation.

2.1 Clearwater: Flexible architectural style for code generation

This section describes architecture of the Clearwater approach and steps involve in the code generation process. The architecture of the Clearwater adopts the complier approach of multiple serial transformation stages which consist of a code generation pipeline. The key idea behind Clearwater approach is that stages typically operates on XML document that is the intermediate representation, and XSLT perform code generation using XSLT transformation. A typical code generation process consist of multiple iterations. The overall code generation process can be summarized as;

Stage1 : Compile Input schemas (SLA, TBL and Workflow) to an intermediate format. This process is a straightforward process and can be considered as transformation step from human readable format into a structural XML.

Stage2 : Pre-processing of XML representation, in the stage extra information is lookup from the disk, resolving names, class paths etc... , and modify XML representation based on the new information.

Stage3 : Code generation using XSLT transformation, which transformation intermediate XML representation into XML + Source code (Language specific and Platform specific). In this stage additional XML tags are also generated alone with the source code to be used in the next step.

Stage4 : Post processing. This step may involve iterative code generation steps that consumes and produced XML elements.

Stage5 : Write generated source code to file system, simply transform XML into Source code.

One of the key advantage of Clearwater approach is, it supports multiple target implementation platforms that requires different types of output. As an example in the case of Rubbos[4] or Rubis[5] benchmark, it generates application sever configuration file, web server configuration file, database configuration file, Windows batch file, and Linux Shell scripts. Furthermore new target output generation can be added just adding new XSLT templates.

2.2 XML : Extensible Language for Domain specification

This section explain the extensibility nature of XML in the context of code generation. The main advantage of XML in the context of code generation approach is its nature of extensibility, additionally simplicity and well-define nature of XML, provide the ability of accepting arbitrary new tags thereby bypassing the overhead encountered when managing both grammar and code generator. As an example there are two ways to add new tags to a XML document. First add new element which enclosed in angle brackets and give new mate-name (e.g., <newelement . . . ></newelement>). Second adding name value pair to an existing element, which is called extending using attributes (e.g., <element newattribute="Value" . . . >). The key benefit is none of the above affect the existing system, unless we make additional changes to XSLT transformation. So that to accommodate the rapid changes in the enterprise distributed application systems, extensibility of XML is a good solution.

2.3 XSLT : Extensible Transformation Language for Code generation

The key benefit of XSLT is to transform XML document into any other document, input to the XSLT transformation always be an XML and out can be either XML or something else. Typically in web application XSLT is used to convert XML into HTML. Each XSLT scripts is a collection of templates, and in the Clearwater approach, each of these corresponds to some unit of transformation from specification or intermediate to intermediate or source code. XSLT transformation has ability to ignore the unknown tags and still generate the valid source code from the input XML, which useful when extending the input specification to cope with system changes. It is the use of XPath [6] and XQuery [7] influence the XSLT with its flexibility; XPath allow a developer to refer to location and groups of locations in an XML tree similar to how a hierarchical file system allows path specification, and XQuery allow to treat XML as a database and execute query against it.

3. CHALLENGES IN STAGING

Staging is the pre-production testing of application configuration with three major goals. First, it verifies functionality, i.e., the system does what it should. Second, it verifies the satisfaction of performance and other quality of service specifications,e.g., whether the allocated hardware resources are adequate. Third, it should also uncover over-provisioned configurations.Large enterprise applications and services are often priced on a resource usage basis. This question involves some trade-off between scalability, unused resources, and cost of evolution . Other benefits of staging, beyond the scope of this report, include the unveiling of other application properties such as its failure modes, rates of failure, degree of administrative attention required, and support for application development and testing in realistic configurations.

These goals lead to some key requirements in the successful staging of an application. First, to verify the correct functionality of deployed software on hardware configuration, the staging environment must reflect the reality of the production environment. Second, to verify performance achievements the workload used in staging must match the service level agreement (SLA) specifications. Third, to uncover potentially wasteful over-provisioning, staging must show the correlation between workload increases and resource utilization level, so an appropriate configuration may be chosen

for production use.

These requirements explain the high costs of a manual approach to staging. It is non-trivial to translate application and workload specifications accurately into actual configurations (requirements 1 and 2). Consequently, it is expensive to explore a wide range of configurations and workloads to understand their correlation (requirement 3). Due to cost limitations, manual staging usually simplifies the application and workload and runs a small number of experiments. Unfortunately, these simplifications also reduce the confidence and validity of staging results.

Large enterprise applications tend to be highly customized "built-to-order" systems due to their sophistication and complexity. While the traditional manual approach may suffice for small-scale or slow-changing applications, built-to-order enterprise applications typically evolve constantly and carry high penalties for any failures or errors. Consequently, it is very important to achieve high confidence during staging, so the production deployment can avoid the many potential problems stemming from complex interactions among the components and resources. To bypass the difficulties of manual staging, Elba advocates an automatic approach for creating and running the experiments to fulfill the above requirements

4. ELBA OVERVIEW

This section discuss architecture and internals of Elba project. The goal of the project is to provide a thorough, low-cost, and automated approach to staging that overcomes the limitations of manual approaches and recaptures the potential value of staging. In Elba processes three major components when automating staging, first the application, second the workload, and third quality of service requirements. One of the main research challenges is the integrated processing of these different specifications through the automated staging steps. In other words we need to merge Service Level Agreement (SLA) [8], Test Based Language (TBL) and workflow into the automating process and come up with a specification for the application. Specification of the production applications and their execution environments consists of a number of research challenges. First, automated re-mapping of deployment locations to staging locations, second creation of consistent staging results across different experiment, third extensibility to many environments and applications.

The overall architectural flow of Elab project for the RUBBoS benchmark is shown in Figure 1, where Elba achieve full automation in system deployment, evaluation, and evolution, by creating code generation tools to link the different steps of deployment, evaluation, reconfiguration, and redesign in the application deployment life cycle. A cyclical view of staging allows feedback from execution to influence design decisions before going to production. The figure shows how new and existing design tools can be incorporated in the staging process if their specification data can be transformed to support staging.

1. Developers provide design-level specifications , Test plan and service agreement
2. Mulini creates a provisioning and deployment plan for the application (configuration , scripts etc..)
3. Deploy the generated client and server application.
4. Execute the staging
5. Collect and Monitor the execution results
6. Manually Analise the date to to figure out what to change to get the expected output.
7. Change the input configurations to meet the expectation.

5. MULINI

In this section we discuss the concepts and architecture of Mulini code generator. In a nutshell Mulini maps a high-level specifications (SLA, TBL and Workflows) of the staging process to low-level tools and code that implement the staging process. Specifications documents are an in-progress language, and those current incarnation is an XML format. Eventually, one or more human-friendly, non-XML formats such as GUI tools or script-like languages will be formulated, and subsequently XTBL will be created automatically from those representations.

The use of XML as a syntax vehicle for the code generator stems from our experiences building code generators around the Clearwater code generation approach. If using traditional code generation techniques that require grammar specification and parser creation, a domain specific language might require a great deal of maintenance with each language change. Experience with Clearwater generators for problems in distributed information flow and in automatic, constrained deployment of applications has shown these generators to be very flexible with respect to changing input languages and very extensible in their support of new features at the specification level and at the implementation level.

5.1 Code Generation in Mulini

As mentioned earlier, the staging phase for an application requires three separate steps: design, deployment, and execution. Again, requirements for automated design are fulfilled by Quartermaster/Cauldron and deployment is fulfilled by ACCT (Automated Composable Code Translator).Mulinis design wraps the third step, execution, with deployment to provide an automated approach to staging. Mulini has four distinct phases of code generation: specification integration, code generation, code weaving, and output. In the current, early version, these stages are at varying levels of feature-completeness. Because of this, we will describe all features to be included in near term releases, and then at the end of this section we will briefly describe our current implementation status.

In specification integration, Mulini accepts as input an XML document that contains the basic descriptors of the staging parameters. This step allows Mulini to make policy-level adjustments to specifications before their processing by policy driven tools such as ACCT+S (SmartFrog). The document, XTBL, contains three types of information: the target staging environment deployment information to which should

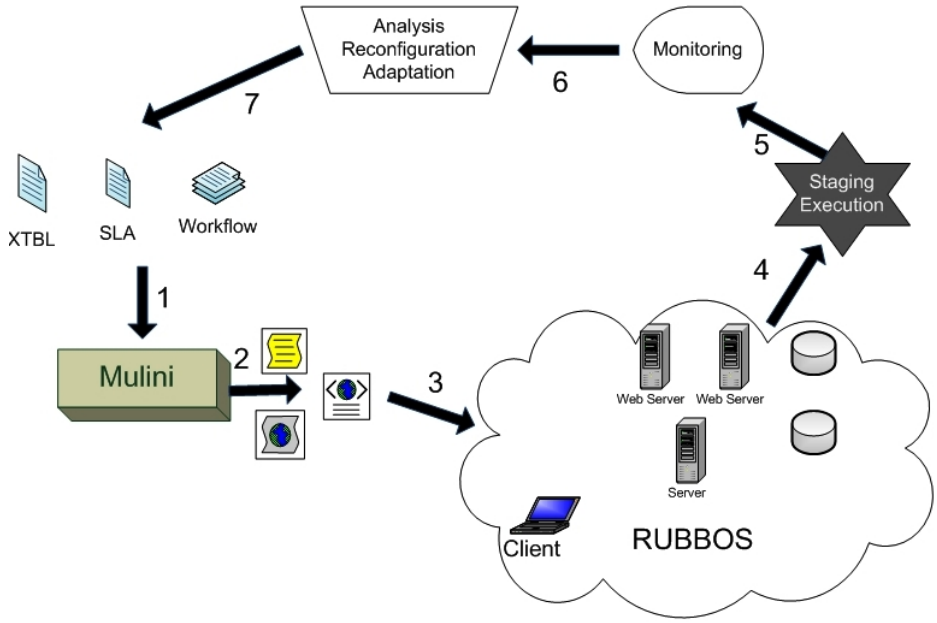

Figure 1: The architecture of Elba project and Staging process with Mulini code generator (for RUBBoS benchmark)

be re-mapped, a reference to a deployment document containing process dependencies, and references to performance policy documents containing performance goals.

5.2 Implementation Based on RUBBoS and RUBiS

RUBBoS is a three-tier e-commerce system modeled on bulletin board news sites similar to Slashdot. The system is known to place a high workload on the database tier. Users are able to perform actions, such as register, view story, and post comment comprising 24 different interactions. The benchmark provides two different client driver modes called browse only and read/write.

RUBiSis a representative application server intensive N-tier application benchmark, and we have chosen an implementation consisting of web server, web container, EJB container, and database server. The application is modeled on online auction sites similar to eBay, and the standard RUBiS client provides two interaction mixes (i.e., browse only and bidding with 15 percent read/write interactions).

Through their real world modeling origins, RUBBoS and RUBiS enable the study of complex distributed system workloads with actual production system significance. In general, the benchmark systems, the software, and the hardware are dependent on a wide range of configurable settings. This flexibility makes accurate prediction of system behavior a non-trivial task and fosters the need for large observational studies. To ensure result reliability and enhance reproducibility, all configurations are kept as close to default as possible.

The first step of the implementation is to configure the Mulini code generators' input schema with the RUBBoS specific configuration. Which includes locations of application softwares (e.g. Apache Server, Tomcat, MySQL, RUBBoS etc ...), output locations (in our case we run the experiment in Emulab [9], so the output location is emulab) , results generation directory, benchmark configurations (number of clients, workload configurations, number of concurrent users). A typical configuration for Mulini code generator for RUBBoS would look like as shown in Figure 2. Next step is to feed the configuration XML to the Mulini code generator and generate all the deployment configurations, deployment artifacts, run scripts, client configurations and client scripts. Once Mulini generate the deployment artifacts and configurations, running benchmark is just a matter of invoking the run scripts which will drive all the other scripts.

```
<?xml version="1.0" ?>
- <xtbl name="RubbosBenchmark" version="0.1">
  - <instances>
    - <params>
      - <env>
          <!-- Experiment name on Emulab -->
          <param name="EMULAB_EXPERIMENT_NAME" value="mulini-yasu6.infosphere.emulab.net" />
          <!-- Directories from which files are copied -->
          <param name="WORK_HOME" value="/proj/Infosphere/yasu/rubbos" />
          <param name="OUTPUT_HOME" value="/proj/Infosphere/yasu/rubbos/Junhee_1_3_1_8ML_ALL" />
          <param name="SOFTWARE_HOME" value="/mnt/softwares" />
          <!-- Output directory for results of RUBBoS benchmark -->
          <param name="RUBBOS_RESULTS_HOST" value="rohan18.cc.gatech.edu" />
          <param name="RUBBOS_RESULTS_DIR_BASE" value="/net/hp48/kanemasa/results" />
          <!-- Target directories -->
          <param name="ELBA_TOP" value="/mnt/elba" />
          <param name="RUBBOS_TOP" value="$ELBA_TOP/rubbos" />
          <param name="TMP_RESULTS_DIR_BASE" value="/mnt/elba/rubbos/results" />
          <param name="RUBBOS_HOME" value="/mnt/elba/rubbos/RUBBoS" />
          <param name="SYSSTAT_HOME" value="/mnt/elba/rubbos/sysstat-7.0.0" />
          <param name="HTTPD_HOME" value="/mnt/elba/rubbos/apache2" />
          <param name="HTTPD_INSTALL_FILES" value="$RUBBOS_TOP/httpd-2.0.54" />
          <param name="MOD_JK_INSTALL_FILES" value="$RUBBOS_TOP/jakarta-tomcat-connectors-1.2.15-src" />
          <param name="CATALINA_HOME" value="/mnt/elba/rubbos/apache-tomcat-5.5.17" />
          <param name="CATALINA_BASE" value="$CATALINA_HOME" />
          <param name="CJDBC_HOME" value="$RUBBOS_TOP/c-jdbc-2.0.2-bin" />
          <param name="MYSQL_HOME" value="$RUBBOS_TOP/mysql-5.0.51a-linux-i686-glibc23" />
          <param name="JONAS_ROOT" value="$RUBBOS_TOP/JONAS_4_6_6" />
```

Figure 2: Mulini input schema for RUBBoS benchmark

5.3 Implementation evaluation

In this section I will discuss how to run the experiment and then analyze data and change the configurations and re-run it. As we discussed in the previous section first step of the experiment is to create the Mulini configuration for the RUBBoS and then generate the deployment artifacts and configurations. First experiment in the series is 1- Web server, 2-Application server, 1-CDJBC and 1-MySQL server. Then we run the experiment and collect the results, then we analyze the results and change the configurations. As we can see in the Figure 3 when the number of concurrent users become 4 with the above configurations response time goes to infinity. Then we change our configurations to 1- Web server, 2-Application server, 1-CDJBC and 2-MySQL and regenerate the deployment artifacts and run the experiment again. Next we analyze the results and reconfigure, likewise until we get the expected response time (according to the SLA) continue the experiment with different configurations.

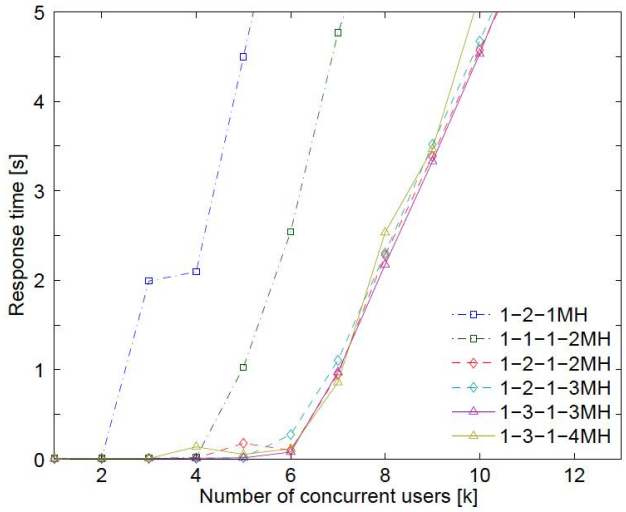

Figure 3: RUBBoS response time changes with number of concurrent users

In our evaluation, we have created a powerful infrastructure to generate the full set of experimental specifications to measure the performance of standard benchmarks over a wide range of hardware and software configurations. We have decided to use this infrastructure to study experimentally the performance variations of these benchmarks over a range of different configurations. Without our code generation infrastructure (Mulini), past performance studies have been limited in scope due to practical problems of managing the number of experiments. We have used the Mulini code generator to create a large number of performance measurement experiments, run the experiments and collect/analyze data automatically, and used the analysis to generate Performance Maps.

6. RELATED WORK

There are few other projects address issues of application monitoring of running applications. Antonia Bertolino et all discussed about Puppet (Pick UP Performance Evaluation Test-bed) [10], an approach for the automatic generation of test-beds to empirically evaluate different QoS features of a Web Service under development. Specifically, the generation exploits the information about the coordinating scenario, the service description and the specification of the agreements that the roles will abide. The approach is supported by a proof-of-concept tool to validate the feasibility of the idea. One of the main difference between their approach and Elba is, Elba trying to cover a board ares however they have only focus on Web service.

One other related approach for SLA monitoring is Dubusman, Schmid, and Kroeger instrument CIM-specified enterprise Java beans using the JMX framework[11]. Their instrumentation then provides feedback during the execution of the application for comparing run-time application performance to the service level agreement guarantees. In the Elba project, the primary concern is the process, staging, and follow-on data analysis that allows the application provider to confirm before deployment that the application will fulfill SLAs. Moreover, this automated staging process allows the application provider to explore the performance space and resource usage of the application on available hardware.

Several other papers have examined the performance characteristics and attempted to characterize the bottlenecks of the Rubbos and Rubis application. These studies generally focused on the effects of tuning various parameters, or on the bottleneck detection process itself . The paper discuss those tools not as the benchmarks, however, but as representative applications which allows us to illustrate the advantages of tuning applications through an automated process with feedback. Most closely related to the architecture of our code generator is that it adopts a similar architecture already used by compilers. Also, it adopts an intermediate format for flexibility like gcc and Flick [12]. However, there are several important features. Traditional compilers only map into basic assembly code. Flick, too, is restricted in its ability to output because it does not maintain a system state document as we do with XIP. This is crucial in achieving the flexibility to do code weaving.

There are also projects such as SoftArch/MTE [13] and Argo/MTE[14] that automatically benchmark various pieces of software Our emphasis, however, is that this benchmarking information can be derived from deployment documents, specifically the SLAs, and then other deployment documents can be used to automate the test and staging process to reduce the overhead of staging applications.

In Apache Axis2 [15] project they also used the XML and XSLT based approach for code generation, however their procedure is very domain specific. Their code generation tool is to generate Web service stubs and skeleton for a give WSDL. Axis2 also follow an approach similar to Clearwater, where code generation consists of number of intermediate steps. Unfortunately in Elba and specifically in Mulin the goal is to generate different type of platform related output.

7. CONCLUSIONS

Currently, there is no reliable way to predict the performance of complex applications (e.g., N-Tier distributed application such as Rubis and Rubbos) in a complex environment(e.g., data centers). The limitations of analytical methods are due to the strong assumptions needed for solving the analytical models (e.g., based on queuing theory)

that are valid only for relatively simple environments. The limitations of experimental measurements are due to the complexity of managing the many configuration combinations in practice. Our work leverages the Elba infrastructure (particularly, the Mulini generator) to generate and manage the experiments, and then use automated analysis techniques and tools to digest the information and create a Performance Map. The Performance Map is a reliable indicator of complex system performance, since it reflects actually measured experiments on the Performance Terrain (modulo tuning and other complications).

8. REFERENCES

[1] Galen S. Swint et al. *Clearwater: Extensible, Flexible, Modular Code Generation*, Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Nov. 2005.

[2] Clark, J.*XSL Transformations*,World Wide Web Consortium (W3C), 1999; http://www.w3.org/TR/xslt.

[3] Bray, T et al. *Extensible Markup Language*,World Wide Web Consortium (W3C), 2004, http://www.w3.org/TR/xml11.

[4] *RUBBoS: Bulletin Board Benchmark*,http://jmob.objectweb.org/rubbos.html.

[5] *RUBiS: Rice University Bidding System*,http://rubis.objectweb.org/.

[6] Sven Groppe and Stefan Bottcher.*XPath query transformation based on XSLT stylesheets*,Workshop On Web Information And Data Management, 2003, pp. 106 - 110.

[7] Zhen Hua Liu, Agnuel Novoselsky.*Efficient XSLT processing in relational database system*,VLDB '06: Proceedings of the 32nd international conference on Very large data bases , 2006.

[8] Joel Sommers et al.*Accurate and efficient SLA compliance monitoring*,ACM SIGCOMM Computer Communication Review, 2007.

[9] *Emulab - Network Emulation Testbed Home*,http://www.emulab.net/.

[10] Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini.*Automatic Generation of Test-beds for Pre-Deployment QoS Evaluation of Web Services*,Workshop on Software and Performance, 2007.

[11] Debusman, M, M. Schmid, and R. Kroeger.*Generic Performance Instrumentation of EJB Applications for Service-level Management*, NOMS, 2002.

[12] Eide, E et al.*Flick: a flexible, optimizing IDL compiler*,In Proceedings of the 1997 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97), 1997.

[13] Grundy, J, Cai, Y, and Liu, A.*SoftArch/MTE: generating distributed system test-beds from high-level software architecture descriptions*,The 16th IEEE Conference on Automated Software Engineering, 2001.

[14] Cai, Y, Grundy, J, and Hosking, J.*Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool*.

[15] Deepal Jayasinghe et al. *Axis2, Middleware for Next Generation Web Services*,CWS '06: Proceedings of the IEEE International Conference on Web Services , Sep. 2006.