Triana: A Graphical Web Service Composition and Execution Toolkit

Shalil Majithia, Matthew Shields, Ian Taylor, Ian Wang
Cardiff School of Computer Science, Cardiff University, Cardiff, UK
{shalil.majithia, m.s.shields, i.j.taylor, i.n.wang}@cs.cardiff.ac.uk

Abstract

Service composition refers to the aggregation of services to build complex applications to achieve client requirements. It is an important challenge to make it possible for users to construct complex workflows transparently and thereby insulating them from the complexity of interacting with numerous heterogeneous services. We present an extension to the Triana PSE to facilitate graphical Web service discovery, composition and invocation. Our framework has several novel features which distinguish it from other work in this area. First, users can graphically create complex service compositions. Second, Triana allows the user to share the composite service as a BPEL4WS graph or expose it as a service in a one-click manner. Third, Triana allows the user to easily carry out "what-if" analysis by altering existing workflows. Fourth, Triana allows the user to record provenance data for a workflow. Finally, our framework allows the user to execute the composed graph on a Grid or P2P network. Triana is a part of the GridLab and GridOneD projects and is used in the GEO 600 project .

1. Introduction

Web services are emerging as an important paradigm for distributed computing [1,2]. It is argued that complex applications will no longer be written as monolithic code but instead be composed from a set of appropriate Web services [3,4]. Current composition frameworks require the user to generate flow specifications which are then submitted to a workflow execution engine. The flow specification details the services being composed and the order of messages that have to be exchanged between these services. There are many Web services flow specification languages like BPEL4WS [9] and WSCI [14]. Similarly, several workflow execution engines are available [15,16,17]. These provide some level of automation in the execution process. However, the use of these frameworks requires the user to do low-level programming. Further, the user is expected to have an understanding of the component services. Specifically, it is assumed that the user is aware of the service required and ensure semantic and data-type compatibility of the messages being passed between the services. Finally, the centralized execution paradigm of these tools makes them susceptible to a single point-of-failure. These complexities make it imperative to provide a composition and execution framework that can make it easy for users to create and execute workflows.

Motivated by these concerns, we have extended the Triana PSE [5,6,7] to allow the graphical composition and distributed execution of services. The Triana framework, as extended, has a number of important features:

- Easy composition of Web services: A major issue in the composition of Web services is the ease with which a user can construct a composite service. Triana allows the user to discover, compose, invoke, and publish atomic and composite services in a transparent manner.
- Distributed execution of composite services: Triana allows the user to execute a composite service on a P2P or Grid middleware.
- Sensitivity Analysis: Triana allows the user to easily carry out "what-if" analysis.
- Recording provenance: Triana allows the user to annotate workflows as well as automatically record provenance related information.

The remainder of the paper is organized as follows: Section 2 provides an overview of Triana and the GAP API. Section 3 describes the components of the WServe API. Section 4 outlines the distributed execution model. Section 5 presents a case study application: a Galaxy formation visualization

application and Section 6 concludes and outlines current status and planned future work.

2. System Overview

In this section, we provide an overview of the Triana system. We outline the general requirements for a web service composition system. We discuss the architecture of Triana, particularly the Gap Interface and the Web Service binding for GAP.

The over-arching requirement is to make it possible for users to construct complex workflows graphically and transparently and thereby insulating them from the complexity of interacting with numerous heterogeneous services. Based on this, a Web service composition system needs the following mechanisms:

- Service discovery methods: there must be a mechanism by which a user, and other components, can locate relevant services on the network;
- Service composition methods: there must be mechanisms to allow composition of services in a simple manner;
- Transparent execution methods: there must be mechanisms to invoke services transparently, i.e. without requiring the user intervention;
- Transparent publishing of services: there must be mechanisms to allow users to publish an atomic or a composite service.

These mechanisms are the basic requirements of a framework for building complex distributed Web service based systems. By facilitating the transparent construction of Web services workflows, users can:

- Focus on the design of the workflow at a conceptual level
- Easily create new complex composite services which offer more functionality than available atomic services;
- Easily expose composite services enabling users to share and replicate experiments;
- Easily carry out 'what-if' analysis by altering existing workflows;

Triana is an open source, distributed, platform independent Problem Solving Environment (PSE) written in the Java programming language. A PSE is a complete, integrated computing environment for composing, compiling, and running applications in a specific application domain [8]. Triana was originally developed for the GEO600 gravitational project [10]. Here, it is used as a data analysis system for analyzing gravitational wave signals that are output from the laser
interferometer detector located in Germany. During the past two years it has been restructured and redesigned in this new federated architecture.

The Triana PSE is a graphical interactive environment that allows users to compose applications and specify their distributed behavior (see Figure 1). A user creates a workflow by dragging the desired units from a toolbox onto the workspace and interconnects them by dragging a cable between them.

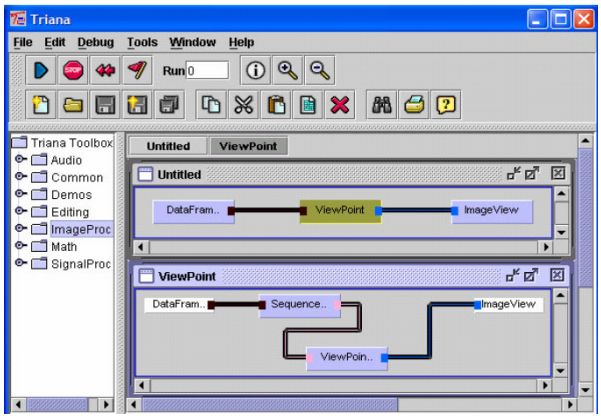

Figure 1: the Triana user interface

Triana has three distinct components: a Triana Service (TS), a Triana Controller Service (TCS) and the Triana User Interface (TGUI) (see Figure 2). TGUI is a user interface to a TCS. The TGUI provides access to a network of computers running Triana service daemons via a gateway Triana service daemon and allows the user to describe the kind of functionality required of Triana: the module deployment model and data stream pathways. The TCS is a persistent service which controls the execution of the given Triana network. It can choose to run the network itself or distribute it to any available TS. Therefore, a single TGUI can control multiple Triana networks deployed over multiple CPU resources. A Triana Service is comprised of three components: a client, a server and a command service controller. The client of the TCS in contact with the TGUI creates a distributed task graph that pipes modules, programs and data to the Triana service daemons. These Triana services simply act in server mode to execute the byte code and pipe data to others. Triana services can pass data to each other also. Clients (i.e. those running a GUI) can log into a Triana Controlling Service, remotely build and run a Triana network and visualize the result (using a graphing unit) on their device even though the visualization unit itself is run remotely.

Users can also log off without stopping the execution of the network and then log in at a later stage to view the progress.

Triana has a pluggable architecture and can be extended in the following ways:

- By using the Triana GUI as a front end to a standalone application. This can be accomplished either by using one of the provided task-graph and command writ-ers or by implementing others. Task graph writers currently implemented include: Petri net, BPEL4WS [6] and a proprietary Triana XML format. The command writer outputs action commands made by the user during an interactive session e.g. play, stop etc. The combination of these allows simulation of user commands from the GUI allowing Triana to run the stand-alone application directly.
- By implementing the same interfaces that the included task graph readers use in the Triana Controller Service. Another distribution and execution mechanism could be inserted with its own resource management and scheduling but taking advantage of the remote log-in features and user interface of the distributed GUI and subsystem.
- By implementing the desired functionality of a specific application within the Triana framework as a collection of Triana units. This would involve decomposing the original application to build a number of Triana components that could be combined to provide the original functionality. This approach can take advantage of the services that Triana provides including type checking, the use of many pre-written Triana units and the seamless distribution of networks using many distribution policies over various types of middleware such as JXTA [13] and OGSA [12].

2.1 GAP Interface

Triana communicates with services through the GAP Interface. The GAP Interface provides applications with methods for advertising, locating, and communicating with other services. The GAP Interface is, as its name suggests, only an interface, and therefore is not tied to any particular middleware implementation. This provides the obvious benefit that an application written using the GAP Interface can be deployed on any middleware for which there is a GAP binding, as shown in Figure 3. This also means that as new middleware, such as OGSA, becomes available, GAP based applications can seamlessly be migrated to operate in the new environment.

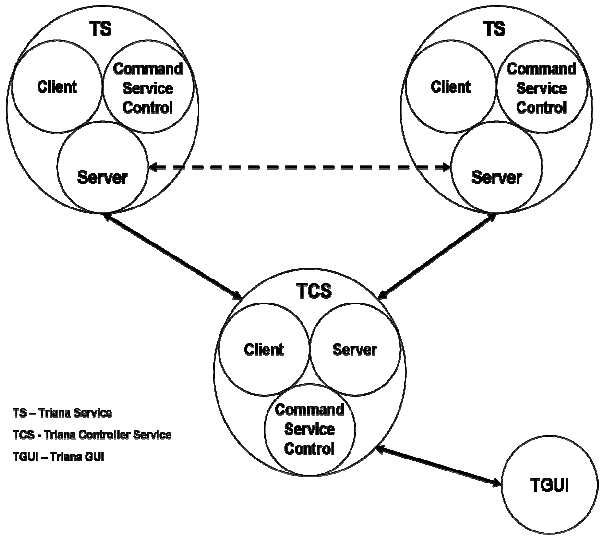

Figure 2. Triana Architecture

Currently there are three middleware bindings implemented for the GAP Interface: JXTA, a peer-to-peer toolkit originally developed by Sun Microsystems [13]; P2PS, a locally developed lightweight alternative to JXTA; and a Web services binding, the implementation of which is discussed in this paper.

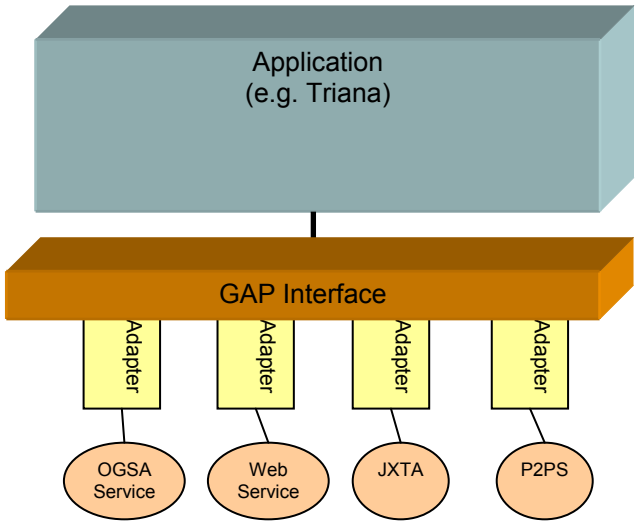

Figure 3: The relationship between Triana, the GAP Interface, and GAP bindings

As mentioned earlier, the GAP Interface provides methods for advertising, locating and communicating

with other peers/services; however the mechanism used to provide this functionality depends on the individual bindings. For example, in the JXTA binding, service discovery is done through the JXTA Discovery Service, while in the Web Services binding, discovery is done using UDDI. As well as providing sufficient functionality in itself to enable the construction of peer-to-peer applications, the GAP Interface was also designed as a prototype for the peer-to-peer subset of the GridLab GAT-API. GridLab is a pan-European project that is developing an easy-to-use, flexible, generic and modular Grid Application Toolkit (GAT) [11]; Triana is a test application for this project. When released, the GridLab GAT will provide a consistent API to resource brokering, monitoring and other services through an adapter architecture. It is intended to implement a GAP-GAT adapter enabling GAP and its bindings to provide peer-to-peer services to GridLab GAT applications.

2.2 Web Services Binding

WServe is an API that implements the GAP binding for Web services (Figure 4). Specifically, it provides the functionality needed to discover, invoke, and publish services. Services are discovered by querying UDDI based on user specified attributes. Services are invoked through a gateway that also handles data type conversions. Composite services can be published to the network by executing a simple GUI based wizard.

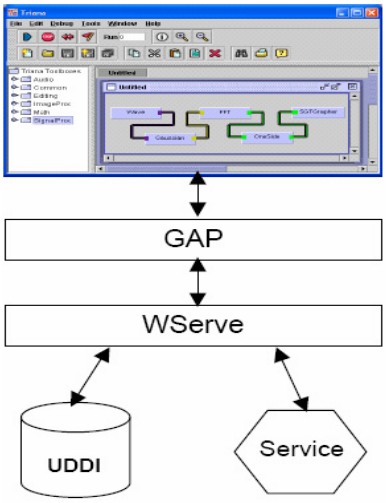

Figure 4: The relationship between GAP and WServe

3. Composition

In this section, we outline how Triana handles service composition. Specifically, we look at the mechanisms provided by the WServe API for service discovery, connecting services, handling data-type translations, and publishing atomic and composite service.

3.1 Discovery of services

Triana allows the user to use services in two ways: query a UDDI or import a WSDL document. The user needs to specify the inquiry/publish address for a public or a private UDDI registry. The discovery module within Triana can then query the UDDI based on simple keyword search. All services matching the keyword are retrieved. WServe provides classes that will search the registry, read the WSDL location address, and retrieve the WSDL document. Triana also allows the user to import a Web service directly by specifying the WSDL location. This could be useful when services are not published in UDDI. The WSDL document is then parsed and a Triana tool representing the service is instantiated. This tool is a proxy for the Web service. On one hand, it interacts with other tools in the composed graph and on the other hand it interacts with the WS Gateway to make the service calls. Interaction with other tools may be receiving or passing data, or control instructions e.g. for looping. Finally, the toolbox is populated to indicate to the user that the service is available and can be used. The service discovery mechanism can easily be extended to handle other types of look-up mechanisms.

3.2 Composing Services

A composite service is created simply by dragging the services required from the toolbox onto the canvas and connecting them with pipes. Triana supports looping constructs (e.g. do-while and repeat-until) and logic units (e.g. if, then etc.) that can be used to graphically control the dataflow, just as a programmer would control the flow within a program by writing instructions directly. In this sense, Triana is a graphical programming environment. This composite graph can then be executed or saved in a format for which a writer is available. We have currently implemented writers in a Triana custom format and BPEL4WS. Additionally, it is also possible to read in a workflow which may have been created by another user. Triana can handle any workflow as long as it is written in a format for which a reader is available. We have

currently implemented a Triana custom format reader and a BPEL4WS reader which can be used to read a graph.

Triana performs dynamic run-time type checking on requested connections to ensure data compatibility between components; this information is extracted from the WSDL document. Triana handles complex data-types by providing a data-type conversion tool which automatically generates the stub. This tool can be dragged onto the canvas and connected within an existing workflow to facilitate data-type conversion. Additionally, the user can create and use a custom serializer. This can be done by using the CreateTool Wizard which automates the entire process of creating the serializer. The stubs generated are locally cached to speed up future invocations.

Once a workflow has been created, it can be executed by pressing the start button on the Triana toolbar. All services with zero input nodes are invoked, and their output data piped to the next task in the workflow. Services that require input data are invoked when the data has been received on all the data input nodes. The invocation is done using SOAP over HTTP.

The Triana GUI makes it easy for the user to carry out "what-if" analysis. Users can make changes to the workflow by adding, deleting, or changing the sequence of execution simply by re-arranging the workflow on the canvas. When the user presses the start button, a new plan is generated and executed.

Triana makes it easy to record provenance data for a workflow. Provenance data recorded includes: date and time composed, the date and time of the execution, details of the resource on which a service was executed, and intermediate data products generated.

3.3 Publishing services

A user can make the composed service graph available to others. This is done by executing a simple GUI based wizard which asks the user to provide the relevant information for e.g. network location, a text description etc. All the necessary artifacts are automatically generated and the service published. The "PublishToUDDI" wizard first uploads the graph to the user specified location. A Triana launcher service running at the server generates a Triana tool representing the composite service. Second, the WSDL document for the composite or atomic service is generated based on the WSDL documents of the component services and the information provided by the user. Finally, the service's details are published to the UDDI server by invoking the UDDI's publish API.

Triana also allows the user to expose local Triana tools as atomic Web services. Triana has a wide-spread user community and there are already over 100 Triana tools. These tools, ranging from image to signal processing, were developed by the Triana user community. Unfortunately, these tools cannot be exposed as Web services without undergoing extensive code revisions. To avoid this, we provide a tool wrapper (see Figure 5) that can handle interactions with Triana tools and provides an interface for it to be invoked from a service. The wrapper configures and invokes the tool. The tool applies the algorithm and passes the results back to the wrapper, which then passes them on to the service

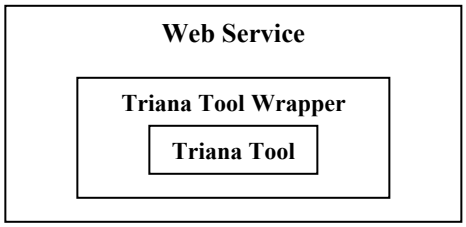

Figure 5: Wrapping Triana Tools

4. Distributed Execution

In this section we examine the mechanisms used by Triana to distribute workflows across a P2P or Grid network. We introduce the concept of Group Tasks and Control Tasks.

4.1 Group and Control Tasks

Group tasks are composite tasks formed by the user by grouping one or more tasks within a workflow. In many ways group tasks behave just like individual tasks: both receive data from a number of input nodes, perform some form of processing on this data, and output the result from their output nodes. Group tasks form the unit of distribution in Triana. If a user wishes to distribute a number of tasks then they create a group from those tasks, and designate that group for distribution through attaching a control task. Group tasks enable the user to determine the granularity of the workflow distribution and which tasks to distribute. Control tasks are standard Triana units that are connected into the workflow to receive the input to a group before it is sent to the tasks within that group, and also to receive the output from a group before it is passed back to the main workflow. Control tasks have two roles: firstly, a control task is responsible for specifying the rewiring of the workflow to support

some distribution topology and secondly, as with looping control tasks, to redirect/manipulate the data input to/output from a group. Between the control task specifying the rewiring of the workflow, and the control task receiving the data input to the group, the Triana engine constructs the specified distributed workflow; thus, when the control task redirects the data it is actually sent to distributed tasks running on remote Triana services. In the following section, we describe how such control tasks annotate the workflow to specify a distribution topology and how they distribute a group's data.

4.2 Workflow Annotation

Each control task is responsible for specifying how the group that it is attached to is distributed. For example, to achieve a parallel distribution the control task would specify that all the tasks within the group are replicated on every service. In Figure 6 we illustrate workflows distributed according to parallel distribution policies.

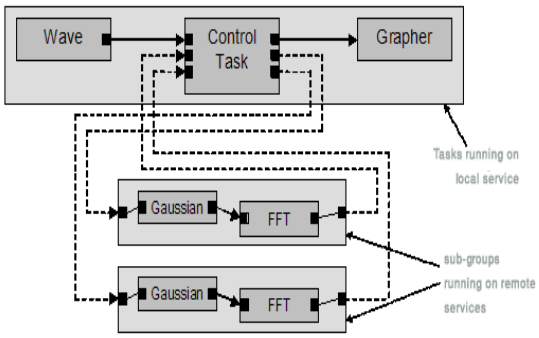

Figure 6: An example of a workflow distributed according to a parallel distribution policy

To create the distribution specification the control task is passed a copy of its group's workflow in an internal representation. From this workflow the control task constructs a number of sub-groups that it requires to be distributed by Triana; with the control task having the freedom to slice the initial workflow however it chooses and to replicate tasks over multiple sub-groups if required. As well as constructing the sub-groups to be distributed, the control task also has to specify how these distributed groups are connected together and back to the local service. To do this the control task creates a unique pipe name for each remote connection, and annotates this name as a parameter in the workflow for both the input and output sub-groups. Thus, when the sub-groups are distributed to remote services the services know the names of the both the input and output pipes they are trying to establish. For each input connection a service simply creates and advertises a pipe of the given name, and the service outputting to that connection locates and binds to that pipe; in this way the distributed workflow is connected together in the topology specified by the control task. Note that the actual mechanism used to create and locate pipes depends on the underlying middleware being used; however, this is not the concern of control tasks. In Triana control tasks are only concerned with specifying the distribution topology and are independent of any particular middleware. The sub-groups created by the control task are passed back to the Triana engine, which is then responsible for locating and setting up a remote Triana service to run each sub-group, and eventually running the distributed workflow.

5. An Example Application: Galaxy Formation

As a test case for the extension, we composed a galaxy-formation visualization application [7] (see Figure 7).

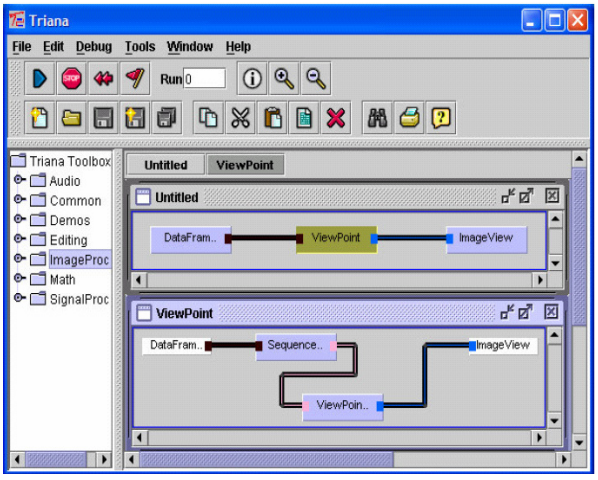

Figure 7: The Galaxy Formation Code implemented as a workflow.

Compact binary stars orbiting each other in a close orbit are among the most powerful sources of gravitational waves. As the orbital radius decreases, a characteristic chirp waveform is produced whose amplitude and frequency increase with time until eventually the two bodies merge together. For this search, we would need around 10 Gigaflops to keep up with the captured data in real time. The signal is divided into data chunks of 15 minutes in duration (around 7.2Mb). This data is transmitted to an

available node where about 10,000 templates are fast correlated with the signal. This procedure is identical to the way we split up the galaxy formation codes, although the data is not. Galaxy and Star formation using smooth-particle hydrodynamics generates large data files containing snapshots of an evolving system stored, typically, in 16 dimensions. After calculation each snapshot is entirely independent of the others with the implication that any data analysis on it can be carried out independently. The independence of frame analysis is an ideal situation for processing data over the Grid. One such analysis is the visualization of particle positions or visualization of line-of-sight mass-density calculation from an arbitrary viewpoint. Both of the calculations require considerable computational resources to create interactive smooth animations, even for modest simulations containing 105 particles. However, by utilizing the Grid, data frames can be distributed and remotely processed, returning a simple small graphic to the client image-viewing and controlling process. These images can then be reassembled in real-time into the correct chronological order to generate a smooth animation. The images in Figure 8 are fairly typical, the calculation that provided these is the result of evolving 120000 sample points: 40000 gas, 40000 Baryonic matter, 40000 star forming particles, courtesy A. Nelson, P. Williams and R. Philp, Department of Physics and Astronomy, Cardiff University

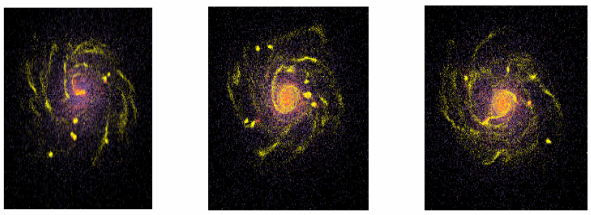

Figure 8: Typical Images as the result of a Galaxy Simulation.

An important point to the user is to be able to visualize the data from any viewpoint interactively. The 3 dimensional data has to be projected to a 2D plane from a particular point in space. This is computationally very expensive, given the number of particles involved. In the application the data file is loaded by a single data reader DataFrameReader unit within Triana and distributed amongst the various Triana servers on the available network. Nodes then buffer the data for future calculations using the Sequence unit. Note that to further increase the speed up the data file could be copied beforehand to the various resources and distributed would then only involve data pointers to the total data set. This is feasible because the simulations that produce the data take a long time to run and the data sets are likely to be well used before new ones are generated. The Sequence unit provides control over the flow of the animation by allowing the user the standard multi-media controls such as play, pause, forward and back. The ViewPointProjection unit's user interface on the user's local machine is used to steer the entire process. The user can select the precise viewpoint by giving the x, y, z coordinates as values or by the slider controls. If the user wants a different view of the data he changes these coordinates and presses the start button. Messages are then sent to all the distributed servers so that the new data slice through each time frame can be calculated and returned. Each distributed Triana service returns its processed data, which is then chronologically re-ordered and used to generate the animation. A Triana visualization unit ImageView then displays the resultant animation as a sequence of GIF files. The result is that the user can visualize the smooth interactive animation of a galaxy's formation in a fraction of the time it takes if it were performed on a single machine.

6. Conclusion

In this paper we have outlined a framework to integrate graphical creation of Web services workflows within the open source Triana problem solving environment. In particular, we looked at how Triana handles discovery, invocation, composition, and publishing of Web services through the WServe API. We have integrated UDDI within Triana enabling us to discover Web services. We have integrated BPEL4WS readers and writers which enables Triana to handle BPEL4WS graphs. Triana can execute a composed workflow or part thereof on a P2P or Grid network. Further, we have developed mechanisms which allow users to graphically and transparently publish atomic and composite services to the network. Current and future work concerns the provision of data provenance, implementation of the complete BPEL4WS specification, integration with other workflow languages like Petri Nets. Additionally, we plan to extend this framework to include the WS-RF Framework. The Triana system is an ongoing project based at Cardiff University; the latest version can be downloaded at http://www.trianacode.org/.

7. References

[1] Fabio Casati, Ming-Chien Shan and D. Georgakopoulos, 2001, "E-Services - Guest editorial." The VLDB Journal 10(1): 1.

[2] A. Tsalgatidou and T. Pilioura, 2002, “An Overview of Standards and Related Technology in Web services.”, Distributed and Parallel Databases, 12(3),
[3] D. Fensel, C. Bussler, Y. Ding, and B. Omelayenko, 2002, “The Web Service Modeling Framework WSMF”, Electronic Commerce Research and Applications, 1(2).
[4] J.Cardoso and A.Sheth, 2002, “Semantic e-Workflow Composition. Technical Report, LSDIS Lab, Computer Science, University of Georgia.
[5] Ian Taylor, Matthew Shields and Ian Wang, 2003, “Resource Management of Triana P2P Services”, Grid Resource Management, edited by Jan Weglarz, Jarek Nabrzyski, Jennifer Schopf and Maciej Stroinski, Kluwer.
[6] Ian Taylor, Matthew Shields, Ian Wang, Roger Philp, 2003, “Grid Enabling Applications Using Triana”, Workshop on Grid Applications and Programming Tools, Seattle. In conjunction with GGF8 jointly organized by: GGF Applications and Testbeds Research Group (APPS-RG) and GGF User Program Development Tools Research Group (UPDT-RG).
[7] Ian Taylor, Matt Shields, Ian Wang and Roger Philp, 2003, “Distributed P2P Computing within Triana: A Galaxy Visualization Test Case”, IPDPS 2003 Conference.
[8] Gallopoulos, E., Houstis, E. N., and Rice, J. R., 1994, “Computer as thinker/doer: Problem solving environments for computational science”, IEEE Comp. Sci. Engr. 1, 11-23.
[9] Satish Thatte, Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Ivana Trickovic, Sanjiva Weerawarana, 2003, “Business Process Execution Language for Web Services Version 1 . 1 .”, h t t p : / / w ww-106.ibm.com/developerworks/webservices/library/ws-bpel/
[10] K. Danzmann and the GEO Team, 1992, “The GEO Project: A Long Baseline Laser Interferometer for the Detection of Gravitational Waves”, Lecture Notes in Physics 410 (1992) 184-209. http://www.geo600.unihannover.de/
[11] G. Allen et al., 2002, “GridLab: Enabling Applications on the Grid”, Grid2002, 3rd International Workshop on Grid Computing, held in conjunction with Supercomputing 2002. Published as LNCS 2002 Vol. 2536, Pages 39-45
[12] I. Foster et al., 2002, “The physiology of the grid: An open grid services architecture for distributed systems integration”, Open Grid Service Infrastructure WG, Global Grid Forum
[13] Project JXTA, 2003, http://www.jxta.org/
[14] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, Sinisa Zimek, 2002, “Web Service Choreography Interface 1.0.” http://www.w3.org/TR/wsci/
[15] IBM BPWS4J http://www.alphaworks.ibm.com/tech/bpws4j
[16] COLLAXA http://www.collaxa.com/
[17] Benatallah, B., Sheng, Q.Z., and Dumas, M.: The Self-Serv Environment for Web Services Composition, Jan/Feb, 2003, IEEE Internet Computing. Vol 7 No 1. pp 40-48.