20

The Triana Workflow Environment: Architecture and Applications

Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison

20.1 Introduction

In this chapter, the Triana workflow environment is described. Triana focuses on supporting services within multiple environments, such as peer-to-peer (P2P) and the Grid, by integrating with various types of *middleware* toolkits. This approach differs from that of the last chapter, which gave an overview of Taverna, a system designed to support scientists using Grid technology to conduct *in silico* experiments in biology. Taverna focuses workflow at the Web services level and addresses concerns of how such services should be presented to its users.

Triana [429] is a workflow environment that consists of an intuitive graphical user interface (GUI) and an underlying subsystem, which allows integration with multiple services and interfaces. The GUI consists of two main sections, as shown in Figure 20.1: a *tool browser*, which employs a conventional file browser interface — the structure representing toolboxes analogous to directories in a standard file browser and the leaves (normally representing files) representing tools; and a *work surface*, which can be used to graphically connect tools to form a data-flow diagram. A user drags a desired tool (or service) from the tool browser, drops it onto the work surface, and connects tools together by dragging from an output port on one tool to an input port on the other, which results in cables being drawn to reflect the resulting data pipeline. Tools can be grouped to create aggregate or compound components (called *Group Units* in Triana) for simplifying the visualization of complex workflows, and groups can contain groups for recursive representation of the workflow.

The underlying subsystem consists of a collection of interfaces that bind to different types of *middleware* and services, including the Grid Application Toolkit (GAT) [13] and, in turn, its multiple bindings to Grid middleware, such as Grid Resource Allocation Manager (GRAM), GridFTP, and GridLab Resource Management System (GRMS); the Grid Application Prototype (GAP) interface [408] and its bindings to JXTA [64], P2PS [460], and

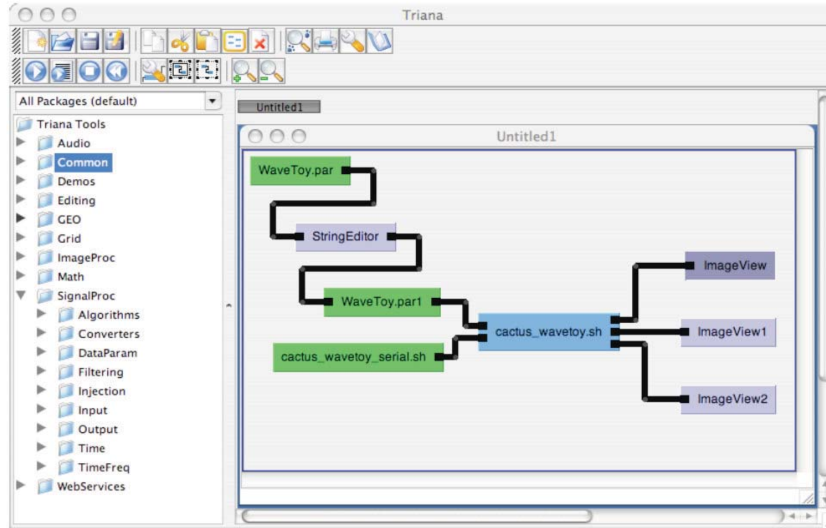

Figure 20.1: A mixed-component Triana workflow consisting of Grid file and job operations through proxies and Java components.

WSPeer [187]; and integration to Web services, WS-RF [100], and OGSA-DAI. The resulting integration means that Triana tools on the work surface can represent any service or primitives exposed by such middleware, and these tools can be interconnected to create mixed-component workflows. An illustration of this is provided in Figure 20.1, where we show a workflow that integrates job and file proxy components that interact with the GAT interface to access job submission (i.e., GRAM) and file transfer operations (GridFTP) and local Java components that provide editing and visualization capabilities. In this example, the local Java components are used to edit a parameter file, which is then staged on the Grid using the file proxy components and specified as an input file for a Grid job submission, which in this case happens to be a Cactus simulation (see Chapter 25). Local Java components are used to visualize the results from this job. Although this example shows the interaction between Grid jobs and local Java units, we have other scenarios that interconnect WS-RF services, P2P services, and local Java units.

In this chapter, we will take a detailed look at the Triana environment and discuss its components for interacting with Grids and P2P networks. We also focus on application examples and describe two specific examples of how workflows are generated, refined, and executed within the environment. The rest of this chapter is organized in the following way. In the next section, we relate Triana to other frameworks described in this book and elsewhere. We then give an overview of the main Triana components and illustrate the types of distributed component interactions that Triana facilitates. In Section

20.5, we discuss the workflow representations Triana uses, and in Section 20.6 how it has been used in a number of different ways by listing some projects that are using Triana and the functionality that they employ. In Sections 20.7 and 20.8, we present two case studies, which illustrate how Triana workflows are generated, modified, and executed within P2P environments and the Grid.

20.2 Relation to Other Frameworks

As we can see from some of the other chapters in this book, the Grid workflow sector is relatively crowded, with a number of different frameworks, languages, and representations for similar concepts. Part of the reason for this is that existing Grid workflow engines are often tied to the technologies employed by their parent projects and are not necessarily able to integrate new technologies effectively. Many of these projects contain elements very similar to Triana, albeit with a different terminology; for example, Triana *tasks* are conceptually the same as Kepler *actors* and Taverna *processors.* The Kepler project (Chapter 7) for supporting scientific workflows is a cross-project collaboration based on the Ptolemy II system [366]. The approach in Kepler/Ptolemy II is very similar to that of Triana in that the workflow is visually constructed from Java components, called actors, which can either be local processes or can invoke remote services such as Web services or a GridFTP transfer.

Taverna (Chapter 19) is a workbench for workflow composition and enactment developed as part of the myGrid [396] project, the focus of which is bioinformatic applications. Originally designed to execute Web service based workflows, Taverna can now interact with arbitrary services. ICENI (Chapter 24) is an environment for constructing applications using a graphical workflow tool together with distributed component repositories on computational Grids. ICENI employs coarser grained components than many of the other environments, generally focusing on large Grid-enabled application components.

The Chimera Virtual Data System (VDS) (Chapter 23) is a system for deriving data rather than generating them explicitly from a workflow. It combines a virtual data catalog, for representing data derivation procedures and derived data, with a virtual data language interpreter that translates user requests into data definition and query operations on the database. The user specifies the desired end result, and a workflow capable of generating that result is derived. If intermediate results are available, then these are used directly rather than being regenerated. Pegasus takes the abstract workflow generated by the Chimera system and maps it onto a Grid. Workflows are expressed in Chimera's Virtual Data Language (VDL) and are converted into Condor's DAGMan format for execution.

The current release of The Globus Alliance's CoG Kit includes a workflow tool called the Karajan Workflow Engine (Chapter 21). The workflow language

Karajan uses is an XML-based scripting language that includes declarative concurrency, support for control structures such as *for…next* and *while* loops, conditionals such as *if…then*, and support for all CoG-supported services, such as GridFTP or Globus job submission. The toolkit comes with a workflow editor for composing Karajan scripts and a workflow engine for executing them. Karajan workflow is aimed specifically at executing jobs in a Grid environment and does not have capabilities for local processes such as those available in Triana's local toolboxes. The main operations with which it concerns itself are job submission and file transfer, and these are represented as nodes in the script. The BPEL4WS (Chapter 14) language is a workflow language for choreographing the interaction between Web services. It is used in many projects in business workflow but is less common in scientific workflow systems.

20.3 Inside The Triana Framework

Triana was initially designed as a quick-look data analysis tool for the GEO 600 project [158] but has been subsequently extended into a number of arenas within the scientific community. Originally, workflows in Triana were constructed from Java tools and executed on the local machine or remotely using RMI. A large suite of over 500 Java tools has been developed, with toolboxes covering problem domains as diverse as signal, image, and audio processing and statistical analysis. More recently, Triana components have evolved into flexible proxies that can represent a number of local and distributed primitives. For example, a Triana unit can represent a Java object, a legacy code, a workflow, a WS-RF, P2P, or Web service, a Grid job, or a local or distributed file.

In essence, Triana is a data-flow system (see Chapter 11) for executing temporal workflows, where cables connecting the units represent the flow of data during execution. Control flow is also supported through special messages that trigger control between units. The cables can be used to represent different functionalities and can provide a convenient method for implementing plug-ins. For example, in our GAT implementation, described in Section 20.4.2, the cables represent GAT invocations, and the content of adjoining units provides the arguments to these calls. Therefore, two connected file units would result in a GAT *fileCopy* invocation, and the actual locations and protocols specified within the units indicate which GAT adapter should be used to make this transfer (e.g., HTTP, GridFTP, and so on).

Triana integrates components and services (see Chapter 12) as Triana units and therefore users visually interact with components that can be connected regardless of their underlying implementation. In a somewhat simplified perspective, Triana *components* are used to specify a part of a system rather than to imply a specific implementation methodology and its obvious object-oriented connotations. Triana components are simply units of

execution with defined interactions, which don't imply any notion of state or defined format for communication.

The representation of a Triana workflow is handled by specific Java reader and writing interfaces, which can be used to support multiple representations through a plug-in mechanism. This means that the actual workflow composition is somewhat independent of workflow language constraints and currently we have implementations for VDL (see Chapter 17) and DAG workflows (see Chapter 22). Such plug-ins can be dynamically activated at runtime, which means that Triana could be used as a translator between such representations to provide syntactic compatibility between systems.

20.4 Distributed Triana Workflows

Triana workflows are comprised of Triana components that accept, process, and output data. A component may be implemented as a Java method call on a local object or as an interface to one of a range of distributed processes or entities such as Grid jobs or Web services. We call components that represent a remote entity *distributed components* without suggesting that the remote entity represented describes itself as a component. These distributed components fall into two categories:

- *Grid-oriented components.* Grid-oriented components represent applications that are executed on the Grid via a Grid resource manager (such as GRAM, GRMS, or Condor/G) and the operations that support these applications, such as file transfer.
- *Service-oriented components.* Service-oriented components represent entities that can be invoked via a network interface, such as Web services or JXTA services.

Triana uses simplified APIs as its base for programming within both service-oriented and Grid-oriented environments. Specifically, the Grid Application Toolkit (GAT) API [13] developed during the GridLab project [175] is used for Grid-oriented components. The GAT is capable of implementing a number of different bindings to different types of middleware, and these can be dynamically switched at runtime to move across heterogeneous Grid environments without changing the application implementation. Section 20.4.2 discusses in detail our core interface to Grid-oriented software toolkits and services using the GAT. For our service-oriented components, we use the Grid Application Prototype (GAP) interface described in the next section. The GAT and GAP interfaces can be used simultaneously within a Triana application instance, enabling users to compose Triana components into workflows that represent elements from both traditional toolkits such as Globus 2.x and Web, WS-RF, or P2P services.

20.4.1 Service-Oriented Components

The Grid Application Prototype Interface (GAP Interface) is a simple interface for advertising and discovering entities within dynamic service-oriented networks. See [408] for a full description of the GAP. Essentially, the GAP uses a P2P-style pipe-based mechanism for communication. The pipe abstraction allows arbitrary protocols to be implemented as bindings to the GAP as long as they can fulfill the basic operations of *publish*, *find*, and *bind*. The GAP currently provides bindings to three different infrastructures:

- *P2PS*. P2PS [460] is lightweight P2P middleware capable of advertisement, discovery, and communication within ad hoc P2P networks. P2PS implements a subset of the functionality of JXTA using the pipe abstraction employed by JXTA but tailored for simplicity, efficiency, and stability.
- *Web services*. This binding allows applications to host and invoke Web services using standard discovery protocols such as UDDI [430] or dynamic P2P oriented discovery mechanisms such as P2PS.
- *JXTA*. JXTA [64] is a set of open protocols for discovery and communication within P2P networks. Originally developed by Sun Microsystems, JXTA is aimed at enabling any connected device, from a PDA to a server, to communicate in a P2P manner.

The GAP abstracts away the implementation detail of the various bindings. For example, service description takes different forms in the existing bindings — Web services use Web Service Definition Language (WSDL) [482], JXTA uses *service descriptors*, and P2PS simply uses named pipes. Likewise, transport and transfer protocols vary — Web services usually use HTTP over TCP/IP, while JXTA and P2PS are transport agnostic, allowing communication to traverse different protocols via the pipe abstraction. These peculiarities do not filter up through the GAP. From a user's perspective, a service is simply made available that provides some capability and can be invoked via the GAP. Furthermore, the use of the GAP as an umbrella to differing service-oriented infrastructures means that it is possible to seamlessly use applications developed on top of the GAP Interface across different networks just by switching the GAP binding used.

The most common GAP binding we use is the Web service binding. This is largely because of its support for Grid-based security, currently via the Grid Security Infrastructure (GSI), and because of the confluence of Web and Grid services, which means many Grid service interfaces are now being defined using Web service standards.

Web Service Integration

The GAP Web service binding is implemented using WSPeer [187]. WSPeer is focused on enabling simple, lightweight Web service management and does not

require the usual infrastructure associated with service hosting, such as a Web server and a service container. Furthermore, it allows an application to expose functionality as a Web service *on the fly*. As a result, WSPeer can operate under diverse conditions, making its binding to a P2P-oriented interface, such as the GAP, a straightforward task.

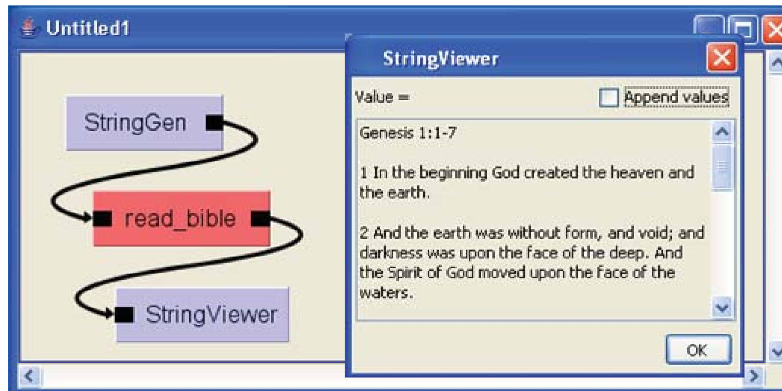

Figure 20.2: Web service and local tools on the Triana desktop.

From the perspective of Triana, the GAP enables diverse service infrastructures to be viewed in a common way. By wrapping a GAP service as a Triana component, the service is made available to the graphical workspace displaying optional input and output ports. Connections are drawn between components with cables, which usually denote data streams. On the workspace, local tools, remote services, and Grid jobs can coexist and be connected to one another. Figure 20.2 shows a combination of local Java tools interacting with a remote Web service. The local tools provide a means for inputting data into and reading output from the service component. The example in Figure 20.2 shows a simple string generator tool that passes Bible book, chapter, and verse information to the service, `read_bible`. The service returns the text from the specified section of the Bible, which is displayed using a simple string viewer tool. From the user's perspective, there is no difference between the components — they are simply visual components on the workspace.

WS-RF Integration

Triana interacts with its distributed resources by using the GAT and the GAP interfaces, which draw a clear distinction between Grid-based

and service-oriented interactions. This distinction divides our distributed interactions between simple application-level interfaces, like the GAT, where clear standardization efforts are currently under way (e.g., the SAGA GGF Research Group [374]) and service-based interfaces. Such a distinction, however, may well become less pronounced as service orientation is more widely adopted by Grid middleware in general.

Therefore, from a Grid service perspective, WSPeer also incorporates Web Service Resource Framework (WS-RF) [100] capabilities that enable Triana to handle stateful resources via services as well as employ the event-driven notification patterns supported by WS-Notification [316].

The WS-RF suite of specifications is based on the concept of a WS-Resource [319]. This is the combination of a resource identifier and an endpoint to a Web service that understands the identifier and can map it to some resource. The resource can be anything — a table in a database, a job, a subscription to a published topic, or a membership in a group of services. The aim of WS-RF is to allow this underlying resource to be made accessible and potentially be modified across multiple message exchanges with a Web service without associating the service itself with the state of the resource. In practice, this is achieved by placing the WS-Resource, serialized as a WS-Addressing [184] EndpointReference, into the header of the Simple Object Access Protocol (SOAP) message. Queries for properties of the underlying resource are implicitly mapped to the resource referenced in the WS-Resource. A WS-RF service advertises the *type* of resource it can handle through a schema document in the WSDL definition of the service. This schema document describes the properties (keys) that the resource type exposes and that can therefore be accessed or modified. When a client is in possession of a WS-Resource, it uses the properties keys declared in the WSDL to retrieve the associated values. These values in turn represent the state of the resource.

Although the underlying infrastructure to manage WS-RF and WS-Notification message exchange patterns is quite complex, Triana makes the process simple from a user's perspective. A WS-RF service can be discovered and imported in the same way ordinary Web services are. When a WS-RF service arrives in the user's toolbox, it is made up of the usual Web service operations that can be dragged onto the Triana worktop to be invoked. However, Triana allows an additional *context* to be associated with these WS-RF service operations in the workflow through a simple GUI. This *context* is not itself a WS-Resource, but the name associated at workflow design time with a WS-Resource that will be created or imported into the workflow at runtime. As WS-RF does not specify the mechanism for how a WS-Resource is created or returned to a client, it is impossible to write an all-purpose tool for creating/importing WS-Resources within a Triana workflow. A typical approach, however, is to employ a factory service. In this case, the factory service can become part of the workflow, feeding the WS-Resource into the context that is used as part of the invocation of a WS-RF service.

WS-RF Workflow

The application of WS-RF compliant services to workflows opens up certain possibilities. In particular, the WS-Resource construct can be used to reduce the need for sending large data sets as SOAP attachments or, worse, encoded as XML. Because the use of WS-Resources allows arbitrary resources to be exposed via a Web service, this can also pertain to data generated by a service (that is, output), allowing a service to return a WS-Resource to a service requester, as opposed to actual data. As a simple example, one can imagine an executable that has been wrapped as a Web service, that takes a file as input, and outputs another file after execution. Using standard Web services mechanisms, one could imagine this service with an operation that takes a byte array, or a SOAP attachment, as input and returns some similar data structure as output. In the case of large files, this can be expensive, especially if the file is being returned to the workflow enactment engine merely to be sent to the next node in the workflow thereafter. If this service is WS-RF compliant, however, then it can return a WS-Resource exposing the file as a resource and itself as the service from which properties of the file can be retrieved. There are a number of ways clients could be given access to the file resource; for example, the resource type may expose a property consisting of a URI that can be connected to directly, in order to read from a data stream. This is far more efficient that transferring data along with the XML and also allows the data to be pulled when (if) needed. If we extend this model to service operation inputs, then it allows us to create workflows in which references to data are passed directly between workflow components, bypassing the need to send data via the enactment engine. Further optimizations can be achieved by defining the properties of the file resource to reflect application-specific requirements. For example, certain services may only need to process parts of the file, in which case a property is exposed that returns just the relevant portion.

From a more general perspective, the widespread adoption of the WS-Addressing specification and the EndpointReference structure is potentially useful in terms of workflow. Although not fully standardized as yet, the use of WS-Addressing could pave the way for a generic means for services to reference each other — similar to the *anchor* tag in HTML — even services that are not specifically Web services. This in turn could lead to mechanisms for describing and deploying heterogeneous workflows — the kind of workflows Triana is capable of building — which are autonomous, running independently of continual controller intervention. Currently, workflows involving arbitrary services still require control and data to pass through the enactment engine at every stage of the workflow because there are no universally accepted means of transferring control or data directly to the next process in the flow.

While the widespread adoption of WS-Addressing should be considered a positive, its use is not always suitable for all situations. In fact, this criticism can be leveled at WS-RF. By combining an endpoint address with a resource

identifier, one is tightly coupling a service with a resource. This model can lead to an object-oriented approach in which WS-Resources are used as global pointers to specific resources located at certain addresses. Furthermore, it encourages the explicit modeling of entities that should be hidden behind the service interface. Both these conditions can lead to complex and fragile systems [261]. Specifically in the context of workflow, managing references that are explicitly tied to service endpoints can become cumbersome as the number of services involved grows. As a result, we are exploring other Web service frameworks, such as the Web Services Composite Application Framework (WS-CAF) [317] and the WS-Context [67] specification in particular, for the generation and enactment of Web service based workflows. WS-Context does not couple state with service endpoints. Instead, context is shared between multiple parties as a stateful conversation, and the interpretation of the context by individual services is left to the invisible implementation of the service.

Web service specification is a rapidly evolving area of development, making it almost impossible to develop code with confidence that it will have any longevity. In fact, we believe it is unlikely that WS-RF will survive in any meaningful form beyond 2007, although some of its ideas may be subsumed into other emerging standards. However, the experience gained with implementing it has left both WSPeer and Triana with flexible architectures for handling contextual message information in general, making them well suited for easily integrating new Web service specifications quickly.

20.4.2 Grid-Oriented Components

Components supporting the execution of code on Grid resources are provided within Triana using the GridLab GAT. The GridLab GAT is a simple API for accessing Grid services and resources. It enables applications to perform Grid tasks such as job submission and file transfer while remaining independent of the Grid middleware used to execute these tasks. The GridLab GAT employs an adapter-based architecture that allows different Grid middleware bindings to be plugged into the GAT, thereby enabling applications written to the GAT API to operate over a range of current and future Grid technologies. The application programmer also benefits from only having to learn a single Grid API, an idea currently being developed further through the SAGA Research Group [374].

At the core of the GAT is the concept of a job, the execution of code on a computational resource. The resource used to execute a job can be local or remote, depending on the GAT adapter used to create the job instance. As essential as job execution is the ability to interact with and relocate files, for example to prestage files in the execution directory of a job and to retrieve output files from a job. Different protocols for accessing and moving files, for example GridFTP and HTTP, can be handled via different GAT adapter instances.

The Visual GAT is the representation of GridLab GAT primitives as components within Triana workflows and the visualization of the data are dependencies between these components. The key Visual GAT components are:

- *Job component.* A job component represents the submission of a GAT job description to a resource broker. This job description includes information on the executable and arguments to be run, plus optional information such as the resource on which the job should be executed.
- *File component.* A file component represents a GAT-accessible file. The file is identified by a URI, which specifies the protocol used to access that file and its network location.

As with standard Triana components, the cables linking Visual GAT components represent data flow between those components. The semantics of this data flow depend on the context of the linked components. For example, the cable between two file components represents data flow from one file location to another; in other words, a file copy operation. Similarly, a cable from a file component to a job component indicates a prestaged file, and a cable from a non-Visual GAT component to a file component indicates a file write.

In Figure 20.1 we show a simple job submission workflow using a mixture of Visual GAT and standard Triana components. In this workflow, local Java components are used to create and view the data, while Visual GAT components are used to represent the prestaging and poststaging of these data and job submission. An equivalent workflow could be created without Visual GAT components; for example, by having specific GridFTP and Globus job submission components. However, although this approach is used within most visual workflow environments, the resulting workflow less accurately models the data flow between workflow components. Furthermore, non-Visual GAT workflows are often more complex and contain more redundancy than equivalent workflows employing Visual GAT components. These issues are discussed in much greater depth in [407].

20.5 Workflow Representation and Generation

A *component* in Triana is the unit of execution. It is the smallest granularity of work that can be executed and typically consists of a single algorithm, process, or service. Component structure in Triana, in common with many component-based systems such as the CCA, has a number of properties such as an identifying name, input and output "ports," a number of optional name/value parameters, and a proxy/reference to the part of the component that will actually be doing the work. In Triana, each component has a definition encoded in XML that specifies the name, input/output specifications, and parameters. The format is similar to WSDL [482], although

more succinct. These definitions are used to represent instance information about a component within the workflow language and component repositories. An example component definition can be seen below.

```
<tool>
   <name>Tangent</name>
   <description>Tangent of the input data</description>
   <inportnum>1</inportnum>
   <outportnum>1</outportnum>
   <input>
      <type>triana.types.GraphType</type>
      <type>triana.types.Const</type>
   </input>
   <output>...</output>
   <parameters>
      <param name="normPhaseReal" value="0.0"
          type="userAccessible"/>
      <param name="normPhaseImag" value="0.0"
          type="userAccessible"/>
   </parameters>
</tool>
```

The external representation of a Triana workflow is a simple XML document consisting of the individual participating component specifications and a list of parent/child relationships representing the connections. Hierarchical groupings are allowed, with subcomponents consisting of a number of assembled components and connections. A simple example taskgraph consisting of just two components can be seen below.

```
<tool>
    <toolname>taskgraph</toolname>
    <tasks>
        <task>
            <toolname>Sqrt</toolname>
            <package>Math.Functions</package>
            <inportnum>1</inportnum>
            <outportnum>1</outportnum>
            <input>
                <type>triana.types.GraphType</type>
                <type>triana.types.Const</type>
            </input>
            <output>...</output>
            <parameters>
            </parameters>
        </task>
        <task>
            <toolname>Cosine</toolname>
            <package>Math.Functions</package>
            ....
```

```
        </task>
        <connections>
            <connection>
                <source taskname="Cosine" node="0" />
                <target taskname="Sqrt" node="0" />
            </connection>
        </connections>
    </tasks>
</tool>
```

Triana can use other external workflow language representations, such as VDL, that are available through “pluggable” language readers and writers. These external workflow representations are mapped to Triana’s internal object representation for execution by Triana. As long as a suitable mapping is available, the external representation will largely be a matter of preference until a standards-based workflow language has been agreed upon. Triana’s XML language is not dissimilar to those used by other projects such as ICENI [153], Taverna/FreeFluo [326], and Ptolemy II [366] and should be interoperable.

A major difference between the Triana workflow language and other languages, such as BPEL4WS, is that our language has no explicit support for control constructs. Loops and execution branching in Triana are handled by specific components; i.e., Triana has a specific loop component that controls repeated execution over a subworkflow and a logical component that controls workflow branching. We believe that this approach is both simpler and more flexible in that it allows for a finer-grained degree of control over these constructs than can be achieved with a simple XML representation. Explicit support for constraint-based loops, such as *while* or an optimization loop, is often needed in scientific workflows but very difficult to represent. A more complicated programming language style representation would allow this but at the cost of ease-of-use considerations.

20.6 Current Triana Applications

This section outlines some of the projects currently using Triana and its related technologies, such as GAP Interface and P2PS. Triana itself is currently being developed as part of the GridOneD project.[1] The GridOneD project is in its second phase of funding. The initial focus of GridOneD was to develop components within Triana to support gravitational wave searches in collaboration with the GEO600 project [158], and this led to the development of GAP, P2PS, and other middleware. The second phase aims to extend Triana’s support for gravitational wave searches and also to develop support for pulsar searches in collaboration with Manchester University and Jodrell

[1] http://www.gridoned.org/.

Bank. This support will employ Visual GAT components within Triana to submit data-analysis jobs across Grid resources.

Triana and its related technologies are being used in a range of external projects. The majority of these projects are using Triana to choreograph Web services. An example of this is Biodiversity World (Chapter 6), a collaboration between Cardiff, Reading, and Southampton universities and the Natural History Museum. The goal of Biodiversity World is to create a Grid-based problem-solving environment for collaborative exploration and analysis of global biodiversity patterns. Triana is providing the visual interface for connecting and enacting the services created by this project. Other examples of projects using Triana to choreograph Web services include Data Mining Grid [107], a project developing tools and services for deploying data-mining applications on the Grid; FAEHIM [8], a second data-mining-based project; and DIPSO [119], an environment for distributed, complex problem solving.

In terms of related technologies, the DARRT (Distributed Audio Rendering using Triana) project [105] at the Louisiana Center for Arts and Technology is exploring the use of Grid computing technologies towards sound computation and music synthesis, in particular using P2P workflow distribution within Triana. The SRSS (Scalable Robust Self-organizing Sensor networks) project [393] has been using the GAP and P2PS in simulating P2P networks within NS2 for researching lightweight discovery mechanisms. Triana is also being used for workflow generation and editing within the GENIUS Grid portal [157], part of the EGEE project.

20.7 Example 1: Distributing GAP Services

The GAP is an interface to a number of distributed services (e.g. P2PS, JXTA, WS-RF, or Web services). Services can be choreographed into Triana workflows for managing the control or data flow and dependencies between distributed services. However, Triana can also be used to locate and utilize a number of distributed *Triana service deployers* by using a distribution policy that enables the dynamic rewiring of the taskgraph at runtime in order to connect to these services. We have implemented two such distribution policies for parallel and pipelined execution. In both scenarios, on the client, a set of Triana units is selected and *grouped* to create a compound unit, and a distribution policy is applied to this group. In the parallel scenario, the subworkflow contained within the group is distributed across all available service deployers in order to duplicate that group capability across the resources. When data arrive they are farmed out to the various distributed services for parallel execution. In the pipelined scenario, the taskgraph is spliced vertically and parts of the group are distributed across the available resources.

These scenarios are based on the P2P-style discovery mechanisms that are exposed by the GAP interface, with implementations of these mechanisms

provided by the different GAP bindings. These scenarios can therefore work over WS-RF and Web services in the same way as for P2PS, as described in Section 20.4.1. We have used this mechanism in a number of scenarios [91, 406, 410, 411] for high-throughput applications, typically on local networks or clusters where we have control of the resources. Each application generally has a fixed set of data that are input into a group unit, which implements a data-processing algorithm, perhaps for searching a parameter space. Typically, the algorithms are CPU intensive and the parameter sets being searched can be divided and sent to parallel instances of the algorithm. We use the parallel distribution policy to discover and distribute the data to available resources for processing.

20.7.1 Workflow Generation

For these types of service-based scenarios, workflows are typically constructed from local units (Java or C) representing the algorithm for importing the data and for performing the parameter search. Such workflows are constructed and prototyped in a serial fashion and then distributed at runtime. The serial version of these algorithms can be complex. In one example [91], a template-matching algorithm for matching inspiral binaries in gravitational wave signals was constructed from more than fifty local Java units with a number of processing pipelines consisting of specific algorithms (e.g., FFT, correlation, complex conjugate, etc.) that were combined and processed further to give the desired result.

Once the algorithm is composed, the user can visually select the *processing* (CPU-intensive) section of the workflow and group it. This group can then be assigned the parallel distribution policy to indicate that it should be task-farmed to available resources. When data arrive at the group unit, they are passed out across the network to the discovered distributed services one data segment at a time. In this way, the individual services can process data segments in parallel.

20.7.2 Workflow Refinement

In this case, workflows are mapped from their locally specified serial version into a distributed workflow that connects to the available resources. This workflow refinement happens at runtime after the client has discovered the available services it can utilize. The workflow is annotated with proxy components to represent the available distributed services, and the workflow is rewired to direct data to these components. This results in the connectivity to the single local group being replaced by one-to-many connectivity from the client to the available remote services.

20.7.3 Workflow Execution

The distributed workflow created during the dynamic refinement process is used by the execution engine in order to be aware of the available services it can use during the execution phase of the workflow. The current algorithm simply passes the data out in parallel to the services and thereafter it passes data to services once they have completed their current data segment. This ensures a simple load-balancing mechanism during execution.

20.8 Example 2: The Visual GAT

In this section, we outline how Triana can be used to implement complex Grid workflows that combine the GridLab GAT capabilities discussed in Section 20.4.2 with interactive legacy application monitoring (using *gridMonSteer*, described in Section 20.8.1). The two scenarios presented below illustrate a fundamental shift in the perception of how legacy applications can be run on the Grid in that the workflow in each example as a whole is the Grid application rather than a monolithic legacy code. The legacy code is typically deployed multiple times within the workflow to conduct parameter sweeps or similar actions, and we allow interactive control in the wider context of the complete workflow.

In the examples presented in this section, we employ the use of a wrapper for integrating distributed legacy applications into complex *Visual GAT* workflows. In these workflows, decisions are made based on the current output state of the legacy application to steer the workflow (or application) to support the appropriate analysis required.

20.8.1 Integrating Legacy Applications

We have implemented a simple, nonintrusive legacy code or application wrapper, called *gridMonSteer* (GMS), which allows us to integrate noncustomized distributed applications within a workflow. GMS monitors the legacy application as it is executing and further allows application and/or workflow-level steering. GMS emerged from an ongoing collaboration investigating the integration of distributed Cactus simulations [167] (Chapter 25 within Triana workflows. Initially, a Grid-friendly Cactus thorn was developed to provide the distributed connectivity from Cactus to a Triana workflow component [168]. This component detected files output by Cactus and passed these into a running Triana workflow, which was used to visualize the simulation as it progressed (this was demonstrated in SuperComputing 2004). GMS is a generalization of this architecture, that allows the same kind of file detection for any application rather than one that is Cactus-specific.

GMS consists of an *application wrapper* that executes a legacy application and monitors specified directories for files that it creates and an *application*

controller, which is any application that exposes a defined Web service interface, enabling it to receive input from one or more application wrappers. The controller, in our case Triana, uses the dynamic deployment capabilities of WSPeer to expose a Web service interface that implements the gridMonSteer protocol for notification and delivery of the distributed files. The wrapper notifies the controller about new files that have been detected. The controller then selects files of interest and returns this list to the wrapper. Thereafter, the wrapper sends these files if and when they are rewritten or updated by the legacy application to the controller. Within the context of the Grid, the wrapper is typically the job submitted to the resource manager, with the executable of the actual legacy application that will be monitored being an argument of this job. Once started, the wrapper executes the legacy application and begins monitoring; for example, in the case of output files, it polls the output directory of the legacy application.

Communication between the wrapper and controller is always initiated by the wrapper. In other words, the controller plays the role of server by opening a listening port and the wrapper that of client in that it opens a per-request outgoing connection, thereby circumventing many NAT and firewall problems that exist within Grid environments. The principal benefit of the gridMonSteer architecture is that the wrapper executes in the same directory as the legacy application, allowing it to constantly monitor the application output files and immediately notify the controller of changes to these files. This approach allows the controller to monitor and respond to intermediate results for the legacy application in a timely manner not possible with other coarse-grained wrapping architectures, such as GEMCLA [229] and SOAPLab [381].

The next two sections describe a brief overview of the two scenarios that use GMS to integrate Cactus within a Triana workflow via the Visual GAT job submission component, described in Section 20.4.2. The breakdown of the process is illustrated through its generation, refinement and execution steps, described in Sections 20.8.4 – 20.8.6.

20.8.2 Executing and Monitoring Dynamic Legacy Applications

This first example was the final project review for the GridLab project. It demonstrated a wrapped GMS Cactus job that was executed within a Triana workflow. Triana was used to stage the files onto the Grid as input to the job, coordinate the job submission, and then interact with the running simulation by visualizing the results and steering it accordingly. The full demonstration is illustrated in Figure 20.3 and is described at length in [407].

Briefly, the scenario involves the following. The *WaveToy_medium.par*, represented using a Visual GAT file component, specifies the location of a Cactus parameter file from a Web server by using an HTTP address. This unit is connected to a local Java component, which results in the HTTP adapter being invoked by the GAT to make the HTTP-to-local transfer. (Conceptually, the GAT invocation is made at the cable level when both protocols on each

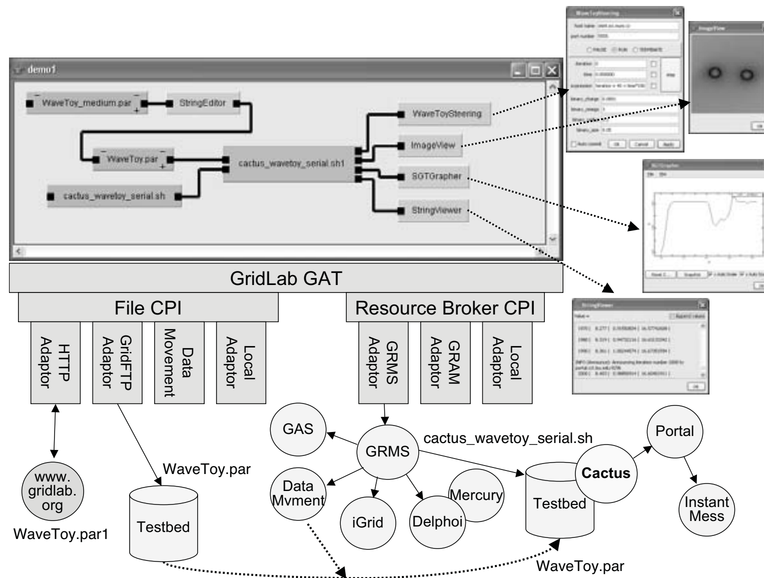

Figure 20.3: Graphical Grid programming through monitoring and steering a deployed Cactus simulation running.

side are defined.) The string editor unit displays this file for minor editing and then passes the contents to another GAT file component that represents a Grid-accessible location (a *gsiftp* address) for the parameter file. The data flow that results used the GridFTP GAT to write the file to a machine in Amsterdam. This file represents the first of two file dependencies for the Cactus job that is specified in the *cactus_wavetoy_serial.sh1* job component. The second dependency is the script that starts Cactus on the remote machine, *cactus_wavetoy_serial.sh*.

The job component in the GridLab review used the GRMS adapter to allow it to make the decision about where the actual job was run. This involved a number of other Gridlab services, including the GridLab Authentication Service (GAS), iGrid, Delphoi, and Mercury, and the GridLab Data Management service, resulting in the *WaveToy.par* and *cactus_wavetoy_serial.sh* files being copied into the location that GRMS chooses to execute the job, as illustrated in Figure 20.3. During execution, we used a custom unit, called *WaveToySteering*, to interact with the Cactus HTTP steering mechanism for changing run-time parameters. We visualized the output files gathered by GMS using the Triana *ImageViewer* tool to display the JPEG files of the simulation, the *SGTGrapher* Triana tool for viewing the wave amplitude of the signal against time from the live Cactus

simulation as it progressed, and the *StringViewer* tool to display standard output (*stdout*) from the simulation.

20.8.3 Dynamic Data-Dependent Simulations

Building from this simple scenario, we are currently in the process of defining more complex Cactus–Triana scenarios that adapt during execution depending on the analyses of data within the workflow. One scenario we are currently implementing involves monitoring a single Cactus simulation much like the scenario above, but instead of steering this Cactus simulation directly, we would monitor its data to watch out for specific features, such as an apparent horizon from a coalescing black hole binary system. Upon such a detection, rather than steering the application directly, we make a decision based on the stimuli or evolution of the application and dynamically instantiate a workflow to aid in the further investigation of this aspect.

Since this typically involves searching a parameter space, we want to perform multiple parallel job submissions of Cacti across the Grid of available resources to perform a distributed search across the parameter range. This could be implemented by dynamically writing a Visual GAT workflow to submit a number of Cacti across the Grid. When the Cacti finish their individual runs, they return the results to the main application, which enables it to steer the main Cactus simulation in the optimal direction and to visualize the results.

20.8.4 Workflow Generation

In both of these cases, the workflows can be specified graphically using simple Visual GAT primitives. In each case, the initial workflows are quite simple and hide the complexity of the multiple levels of refinement that can happen during execution.

20.8.5 Workflow Refinement

In both of these scenarios, Triana can refine the workflows in a number of different ways. In the first case, there are two levels of refinement, which were outlined during the scenario. The first involves converting the abstract Visual GAT workflow into a set of invocations that are appropriate for deployment. We described two mechanisms in the scenario for file transfer and job submission. The virtual GAT invocations result in runtime level refinement by dynamically choosing the appropriate Grid tool for the capability. So, for HTTP-to-local file transfer, an HTTP file adapter was used, and for Grid staging, a GridFTP file adapter was used. Similarly, for job submission, GRMS was chosen for its discovery and resource brokering capabilities.

A second-level refinement was made at application steering by allowing the location of the simulation to be dynamically fed into the *WaveToySteering*

unit, which could, in turn, tune the parameters in the simulation. The application-level refinement allows a user to alter the behavior of the simulation which in the case of the first scenario can result in different internal workflows taking place. In the second scenario, however, this is more apparent. Here, the result from one simulation is used to drive the workflow as a whole. The initial workflow is simple, but as events are detected, more workflows are spawned to analyze these events further and are then fed back into the workflow in order to steer the Cactus simulation.

20.8.6 Workflow Execution

The execution of both of these workflows uses the underlying GAT engine to coordinate the execution of the components and stage the files for the necessary transfer. Triana simply acts as a graphical interface to this underlying engine for the distributed functionality connecting these stages to the default local scheduler for execution of the local units where appropriate. Triana can also mix and match distributed services, local units, and GAT constructs and therefore acts as a manager or a bridge between the different engines for execution of the components.

20.9 Conclusion

In this chapter, we described the Triana workflow environment, which is capable of acting in heterogeneous Grid and P2P environments simultaneously. This is accomplished through the use of two lightweight application-level interfaces, called the GAP and the GAT, that allow integration with distributed services and Grid capabilities. The underlying bindings for these interfaces allow interaction through the GAP to JXTA, P2PS, and WSPeer (with its integration to Web services and WS-RF) and through the GAT to a host of Grid tools, such as GRAM, GridFTP, and GRMS. We described each of these bindings and outlined the underlying workflow language on which Triana is based. Finally, we presented two service-based and Grid-based examples to show how the workflow is generated, refined, and executed in each case.

20.10 Acknowledgments

Triana was originally developed within the GEO 600 project funded by PPARC but recent developments have been supported through GridLab, an EU IST three-year project and GridOneD (PPARC), which has funded Grid and P2P Triana developments for the analysis of one-dimensional astrophysics data sets. GridOneD, initially a three-year project, has recently been renewed for a further two years.