Service-enabling Legacy Applications for the GENIE Project

Sofia Panagiotidi, Jeremy Cohen, John Darlington, Marko Krznarić

London e-Science Centre, Imperial College London, South Kensington Campus, London SW7 2AZ, UK
Email: {sp1003,jhc02,marko,jd}@doc.ic.ac.uk

Abstract. We present work done within the Grid ENabled Integrated Earth system model (GENIE) project to take the original, complex, tightly-coupled Fortran earth modeling application that has been developed by the GENIE team and enable it for execution within a component-based execution environment. Further we have aimed to show that by representing the application as a set of high-level Java Web Service components, execution and management of the application can be made much more flexible. We show how the application has been built into higher-level components and how these have been wrapped within the Java Web Service abstraction. We then look at how these components can be composed into workflows and executed within a Grid environment.

1 Introduction

Modularity and component wise construction is a central feature of all grid systems. Grids provide simplified execution of large-scale scientific applications, across multiple computational resources and organisations. The Grid ENabled Integrated Earth system model (GENIE) is a climate model simulation developed under the NERC-funded GENIE and GENIEfy projects. This computationally intensive application, written in Fortran, is ideally suited to Grid environments. However the existing, tightly-coupled, sequential application model cannot take advantage of the benefits of Grid systems.

We present work undertaken to modify the GENIE model allowing the existing Fortran implementation to take advantage of Grid execution features. GENIE consists of a set of modules simulating various earth entities (e.g. atmosphere, sea-ice etc.). Entities may have multiple module implementations that are coupled to produce a complete representation of an earth simulation. We present an abstract component model used to wrap the existing GENIE modules into higher-level components, taking into account features of the existing framework that present difficulties for traditional component models. Rather than rewriting the application from scratch in a higher-level language such as Java, an unreasonably complex task, we wrap the existing code in Java wrappers and show how the resulting components can be composed into workflows that may be executed through Grid middlewares.

The rest of the paper is as follows: section 2 discusses in more detail the structure of the GENIE application and some basic features of its architecture and implementation. Some previous work is summarised. Section 3 provides the neccessary background on grid-enabled component architectures. In the following section our visionary model for GENIE is presented and explained in detail. In section 5 some issues and technical details on how we have been working towards wrapping up GENIE modules in order to move closer to the goal are discussed. Last, Section 6 concludes and discusses further work.

2 Earth System Models and GENIE

2.1 General Description

GENIE [2] is a scalable modular platform aiming to simulate the long term evolution of the Earth's climate. Building an Earth system model (ESM) involves the coupling of a set of specialised components. Thus, distinct earth entities such as the atmosphere, the ocean, the land, the sea-ice etc., referred to from now on as *earth modules* or just *modules*, are modelled independantly, using specified meshes of data points to represent the boundary surfaces, and

conform to the natural laws of physics that govern the exchange of energy and fluxes from one to another.

One of the architectural characteristics of GENIE is that it is designed so as to consist of more than one implementation (*instance*) of an earth element in the case of the atmosphere, ocean, land and sea-ice. Furthermore, scientists are able to choose to experiment with several simulation scenarios (*configurations*) comprising of combinations of such implementations. The way this was implemented is through a separate program, "genie.F", which is responsible for the control and execution of each module, the attribute passing between the chosen ones and the appropriate interpolations of the data exchanged, through interpolation functions. In fact, all possible cases of configurations are handled through the "genie.F" with the use of flags, being switched on/off to specify the use of which implementation of a module.

As new modules are actively being researched and developed, it is desirable for the GENIE community to have the flexibility to easily add, modify and couple together GENIE modules and experiment with new configurations, without undue programming effort. Despite significant progress in the GENIE community, the desired result is far from reality.

Terms: component, (earth) module, instance, port, grid, wrapper, glue (code), abstract interface, concrete interface

2.2 Previous work

Several milestones towards the advance of the GENIE framework have been achieved. The first task was to separate the tightly coupled Fortran code into distinct pieces each representing an environmental module (atmosphere, land, ocean, ocean-surface, chemistry, sea-ice, etc). The main piece of code handling and coupling the modules was disengaged and formed the so-called "genie.F" top program.

Efforts in gradually moving GENIE towards a grid environment have been made in the past, most importantly allowing the execution and monitoring of many ensemble experiments in parallel [4, 6], reducing the overall execution time. Though, they all deal with a subset of the gridification issues and serve the isolated current at that time needs of the scientific community.

Previous work also includes research into the way the ICENI [3] framework can be used to run parameter sweep experiments across multiple Grid resources [5].

Lastly, there have been efforts into wrapping up each of the basic Earth modules using the JNI [7] library, which led to unexpected complications and unjustified amounts of effort, leading us to look into a more efficient solution.

3 Component-based Frameworks

The basic reason why the design of GENIE has not achieved desired level of modularity, which is the objective, is that it is fundamentally restricted by the use of Fortan as a scientifically efficient language [8].

The component programming model is the latest stage of a natural progression, starting from the monolithic application, and moving towards applications built from increasingly more modularised pieces, with one addition, the coupling framework. Components are designed to be protected from changes in the software environment outside their boundaries.

Thus, a component can be described as a 'black box', fully specified by its function specification and the input/output type patterns of the data (ports). The fact that a component is interfaced with other modules/systems only through its input/output ports (and is otherwise shielded from the rest of the world) allows bigger applications to be designed, implemented, and tested, independently of everything else.

The coupling framework within component architecture is a system defining the bindings between the components in a clear and complete way and providing information about the run-time environment. It can include connectors which perform the direct "binding" of the ports of two components or even more composite operations on the input/output/inout ports. Finally, a configuration of components may be (re)used as an individual component in another (sub) framework. In this context, an application like GENIE may be composed (assembled) at run-time from components selected from a component pool.

An example of an ideal composition environment would be the Imperial College e-Science Networked Infrastructure (ICENI) [3]. ICENI is a pioneering Service Oriented Architecture (SOA) and component framework developed at the London e-Science Centre, Imperial College London. Within the ICENI model an abstract component can have several differing

implementations. Associated with each component is metadata describing its characteristics. At deployment, this metadata is used to achieve the optimal selection of component implementation and execution resource in order to minimise overall execution time given the Grid resources currently available.

4 A Grid Model for GENIE

4.1 Abstract Component Model

Within the ICENI model, semantically equivalent components have the same interface. In fact there need only be one semantically equivalent *abstract* component, concrete implementations inherit from this. In earth modelling systems such as GENIE, however, different implementations of a semantically equivalent earth entity module may have differing interfaces. For example, one ocean model may incorporate salinity while another does not. However, they are both ocean models and should be interchangable. We therefore extend the ICENI philosophy to provide different mechanisms whereby similar models but with differing interfaces can be used in an interchangable fashion. We do this by extending the notion of an interface by wrapping the module with plugin adapters to accomodate the different interfaces. We describe this interface model and show how the implementation can be achieved through Babel.

4.2 Extended Abstract Component Model

To realise this model we define four key questions which we then look at in greater detail:

- How are abstract models linked/composed?
- How is a GENIE system comprising various models assembled?
- How is GENIE system deployed in a distributed dynamic environment?
- How do GENIE models communicate with each other?

In order to compose abstract models in an interchangeable manner, it is necessary for the models to have common interfaces. In the case of GENIE, it is possible to encounter situations where semantically equivalent modules, that should in theory be interchangeable, cannot be substituted due to differing interfaces. We tackle this issue by defining a pluggable abstract interface that allows the knowledge of the module developer to be encapsulated within a 'plugin'.

To build a GENIE system from a given set of modules it is necessary to compose the modules in a semantically valid format. Using component metadata, an idea utilised extensively in ICENI components, we can annotate the GENIE components with information that determines their composability.

By wrapping GENIE components in a Java Web Service wrapper, it is possible to deploy the components in a service container allowing them to communicate using standard Web Service protocols. The use of Web Service standards provides mobility and allows the location of component deployment to be determined at application run time. The deployment of GENIE in a distributed, dynamic environment is discussed in more detail in section 5.

The communication between GENIE models may be accomplished using a variety of methods dependent on where they are deployed. Components deployed on geographically distributed resources may communicate using Simple Object Access Protocol (SOAP) messages over an HTTP connection, although this may not be the most efficient. Components that are co-located within a cluster may use MPI or shared memory to communicate (figure 1).

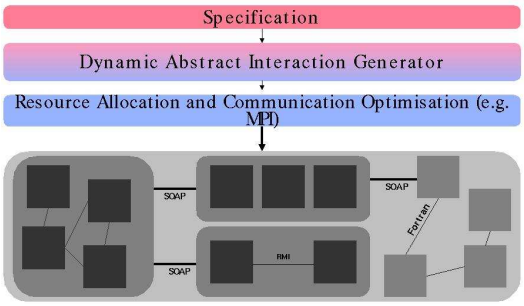

Fig. 1. Grid-based efficient model coupling and deployment

We now describe in more detail the idea of an abstract component that provides a common interface to a set of concrete implementations of a given type of model, even though those implementations may have differing interfaces. Figure 2 shows this model at the simplest level.

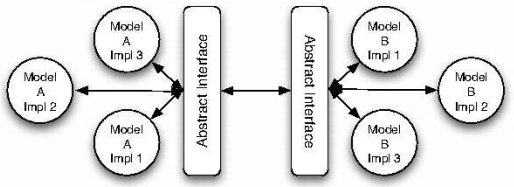

Fig. 2. A model using abstract interfaces to access concrete implementations.

The implementation of this abstract interface uses the idea of transformation plugins. A transformation plugin is a unit of code that is developed by the model developer and encapsulates their knowledge of the input and output format of data that their model accepts or produces. The plugin acts as a translation layer between a more general – although still model specific – interface and the concrete interface provided by the specific model implementation.

Due to the plugin architecture, we consider our abstract component interface to be more of a wrapper for a set of model implementations rather than simply a standard interface. This leads to the more detailed layout shown in figure 3. Our model supports both static, compile time generation of the abstract wrapper for a set of models and also dynamic, runtime registration of new models into the wrapper.

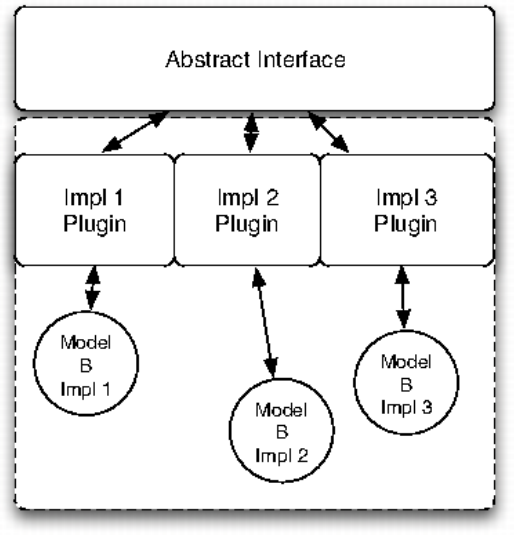

Fig. 3. Abstract interface marshalling requests to correct concrete implementation through plugin translators.

5 Grid Enabling the GENIE Application

5.1 Determining Component Ports

In the GENIE structure, each earth module is represented by a set of files, including Fortran routines, namelists, input files, netCDF data files and others. All these elements are represneted in the GENIE structure separately for each earth entity, to maintain the modularity of the code. The main functionality of each earth module is implemented in one subroutine. This subroutine requires/passes arguments corresponding to the physical quantities that the boundaries of the module receive and process. Furthermore, subroutine files containing initialisation, restart and finalising processes for the module are also part of its code structure. Thus, each physical module has its own hierarchy with a basic routine as well as initialisation/restart/finalisation routines. These routines are considered to be "top-level" as they are the only ones that directly communicate with the GENIE environment through the "genie.F" application controller.

In order to describe and access an earth module through an interface of a high level language, identifying the inputs and outputs is essential. This task, in the case of GENIE, is far from trivial. Although, as mentioned before, the arguments of the top-level routines are those that need to be part of the interface, the separation of these arguments into inputs, outputs and inouts is difficult. The reason for this is that Fortran passes all arguments to a routine by reference. This means that it is very difficult to extract any information about whether a parameter is strictly used as read-only part of the memory, or gets modified during the execution of the specific piece of code.

During our inspection and detailed analysis of the GENIE modules, a long and complex task, the nature and type of the module parameters was documented. This served not only for our purposes, but has provided useful input to the GENIE project for future needs of the scientists and users of the application. In several cases, a manual exhaustive in-depth analysis of the code of a routine, together with the subroutines called within the code needed to be done in order to determine whether an attribute is written to, is of read-only nature, or possibly serves both read/modify purposes.

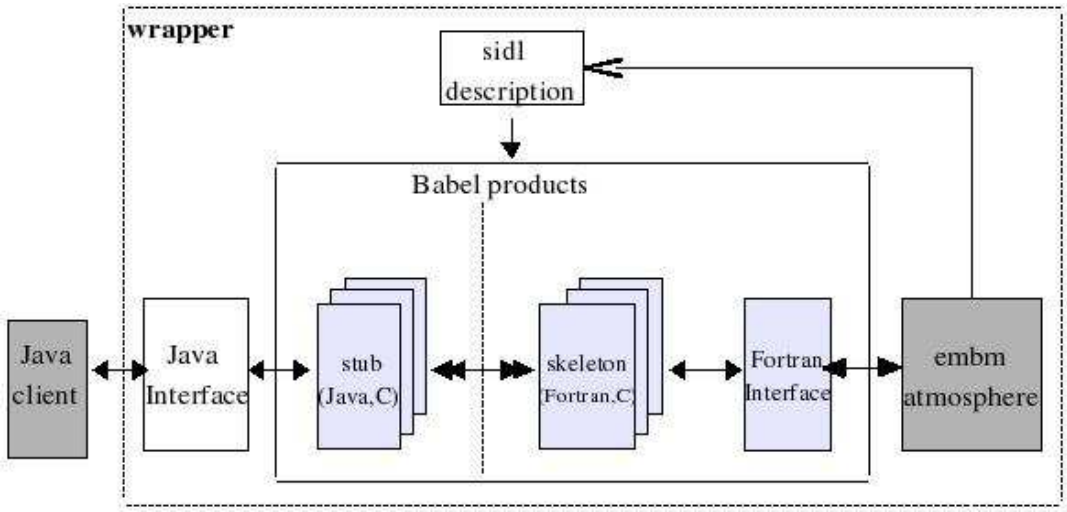

Fig. 4. Wrapping in Babel

5.2 Linking Fortran to Java

In order to form high level components that can be launched and used as web services we choose Java to be the main language in which the interfaces of the earth modules will be exposed.

Rewriting them would possibly cause a number of inaccuracies and most importantly would cause a disruption in the development of such a complex scientific application. Therefore, we choose to wrap each earth module individually using an efficient and suitable mechanism.

For this purpose we pick Babel [1], an interoperability tool which addresses the language interoperability problem in software engineering. Babel uses a prototype SIDL (Scientific Interface Definition Language) interface description which describes a calling interface (but not the implementation) of a particular scientific module. Babel, as a parser tool, uses this interface description to generate glue code (server skeleton and client stub) that allows a software library implemented in one supported language to be called from any other supported language.

Below we denote some of the advantages when using Babel, as against other mechanisms, to wrap up GENIE modules:

- Babel provides all the three kinds of communication ports used by the GENIE subroutines – inputs, outputs, as well as inouts, the latter of which requires extra programming effort to be implemented via other mechanisms (e.g. JNI),
- It minimizes the problems we identified in [8] when using JNI, such as arrays stored differently in memory (C/Fortran), floating point numbers, complex structure argument passing, manual creation of intermediate ".h" files and programming effort.
- Possible future addition of extra components becomes easy, since it provides a standard mechanism for wrapping up components.
- Babel is compatible with most Fortran compilers used by scientists in GENIE.
- An SIDL file can include methods such as the initialise component, main and finalise, making it possible to expose all separately developed subroutines as parts of the same component, as semantically correct and desired, conforming to the component lifecycle of modern component architectures.
- An interoperability tool like Babel is particularly useful to make heterogeneous environments communicate, since our purpose is not only to make GENIE modules available over the web but moreover, to provide a generic methodology independent of the component implementation language.

After testing and verifying its suitability for the project's needs, we have been using Babel to wrap a sample of components and make them accessible from a Java client which is designed to replace "genie.F". Throughout this procedure several complications were encountered but resolved.

Firstly, Babel uses its own defined data types (from here and on referred to as *sidl-types*) in the SIDL interface specification that it accepts, producing code which handles similar data types in the language specified by the user. Thus, the code which is generated in Fortran, designed to access an earth module, needs some additions in order to serve its purpose. For this reason, several external routines "casting" the sidl-types of data from/to Fortran data types were developed. In this way, it was made possible to have a one-to-one correspondence of integers, floating point numbers, as well as one and two dimensional arrays to and from the equivalent sidl-types. Having these casting routines, the glue code can access the top level module routine by passing the arguments and having them returned after the addition of a piece of "casting" code inside the glue. It needs to be mentioned here that in this way, there is absolutely no need to modify any of the existing module code in order to access it from the skeleton created from Babel. Therefore, just by adding an additional structure for each module in the tree code of GENIE containing the SIDL descriptors and the generated glue code, the maintenance of the structure is guaranteed. Furthermore, this does not interfere with any attempt to run the GENIE application without using the wrapped modules and the componentised version, but executing it in the old-fashioned way where "genie.F" handles the modules.

Another feature of Babel is that it works with the use of shared libraries. The code of the module gets compiled together with the glue code to form a library, which then can be placed independently and accessed through a client from anywhere on the web. The GENIE application structure already contains the code of a module in an independent, shared library. Therefore, it is only sufficient to compile to glue code against the module library to ensure the communication of the two, allowing a more distributed environment where the skeleton accesses the actual implementation remotely.

We end up with a module wrapping procedure (figure 4) that enables an earth module (and eventually a whole configuration) to be accessed by a Java client, in the following simple stages, which require minimal programming effort:

- describe the input/output/inout ports of the module top level Fortran subroutine in a simple SIDL file
- run Babel parser to create glue code
- connect the top level subroutine of an earth module to the skeleton by passing the ports to/from Babel skeleton and to/from subroutine

5.3 Executing GENIE in a Grid Environment

So far, we have managed to wrap up two simple GENIE components using Babel and tested the results returned when calling them from a simple Java client. Our current activities mainly include a specific configuration (figure 5) comprising the GENIE basic ocean (Goldstein), a simple atmosphere (Embm) and a simple compatible sea-ice (Goldstein sea-ice). We have isolated this case as the most suitable to study and implement using Web Services. The three earth modules, together with their initialisation, and finalisation routines are being described and accessed through a Java interface and at the same time a Java client is being developed in order to access, and couple the components. Initially the configuration chosen is being tested by direct access of the interfaces exposed by the wrapped modules, without the use of Web Services, for reasons of simplicity.

It needs to be made clear that any component composition environment requires every entity to be in the form of a software component. Thus, all the functionality of "genie.F" must be disguised in a component's structure and become part of the data workflow. For example, all intermediate interpolation functions of data exchanged between modules as well as various functional and non-functional operators being part of the component composition phase, should become part of the (GENIE) application specification phase.

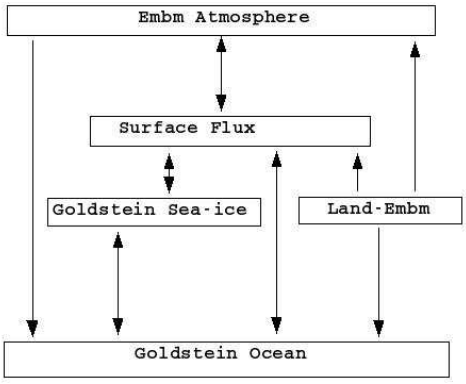

Fig. 5. Configuration to be implemented in Web Services

Below, we summarise the various enginnering steps which are planned to take place in order to achieve the grid enabling of the GENIE application using a Web Service-based workflow within the London e-Science Centre.

1. Wrap up a component in Java using Babel and test it individually to ensure proper results during execution.
2. Wrap up all modules for one existing configuration. Then create a Java client to synchronise the components and compare the results to those produced when executing "genie.F" over the same configuration.
3. Launch Web Services in place of the original components and modify the Java client to call these Web Services instead of the components.
4. Create all possible configurations of GENIE in a similar manner to that shown in the previous steps.
5. Create the Abstract Component Wrapper by choosing carefully what the concrete interface will be.
6. Use a workflow engine to orchestrate the Abstract Components in all configurations.
7. Advance the previous stages into a Grid environment so as to incorporate Coordination Specification, Dynamic Abstract Code Generation, Resource Allocation and Optimisation.

6 Conclusions

We have taken the Fortran implementation of the GENIE earth simulation application and shown how this can be modified to take advantage of a service-based computational Grid execution model. Through analysis of the original, tightly-coupled implementation, the component interfaces were identified. It was then possible to apply an abstract component model to the various earth entity module implementations to provide substitutable components that can be composed into workflows for scheduling and execution on Grid resources. Our implementation takes the form of a Service Oriented Architecture utilising Web Services as the service model. We have shown that it is possible to take a complex legacy application and apply wrapping techniques for service-enabling without the extensive effort that would be required for a rewrite of the original code in a different language. The work provides simplified deployment of GENIE workflows across multiple resources in combination with the opportunity for improved runtimes that distributed Grid execution allows. We intend to continue this work to service enable further modules within the GENIE framework.

7 Acknowledgements

We would like to thank NERC who have funded the GENIE project and its follow up GENIEfy, Tim Lenton and the rest of the GENIE/GENIEfy project team.

References

1. Babel home page. http://www.llnl.gov/casc/components/babel.html.
2. Grid enabled integrated earth system model (genie). http://www.genie.ac.uk.
3. The imperial college e-science networked infrastructure (iceni). available at: http://www.lesc.ic.ac.uk/iceni.
4. M. Y. Gulamali, A. S. McGough, S.J. Newhouse, and J. Darlington. Using iceni to run parameter sweep applications across multiple grid resources. In *Global Grid Forum 10, Case Studies on Grid Applications Workshop*, Berlin, Germany, Mar. 2004.
5. M.Y. Gulamali, T.M. Lenton, A. Yool, A.R. Price, R.J. Marsh, N.R. Edwards, P.J. Valdes, J.L. Wason, S.J. Cox, M. Krznaric, S.J. Newhouse, and J. Darlington. Genie: Delivering e-science to the environmental scientist. In *UK e-Science All Hands Meeting 2003*, pages 145–152, Nottingham, UK, 2003.
6. M.Y. Gulamali, A.S. McGough, R.J. Marsh, N.R. Edwards, T.M. Lenton, P.J. Valdes, S.J. Cox, S.J. and Newhouse, and J. Darlington.

Performance guided scheduling in genie through iceni. In *UK e-Science All Hands Meeting 2004*, pages 792–799, Nottingham, UK, 2004. ISBN=1-904425-21-6.

7. Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 1999. ISBN=0201325772.

8. S. Panagiotidi, E. Katsiri, and J. Darlington. On advanced scientific understanding, model componentisation and coupling in genie. In *All Hands Meeting, Nottingham*, September 2005.