Meaning and Behaviour in Grid Oriented Components

Anthony Mayer, Stephen McGough, Murtaza Gulamali,
Laurie Young, Jim Stanton, Steven Newhouse, and John Darlington

London e-Science Centre, Imperial College of Science, Technology and Medicine,
London, SW7 2BZ, UK
icpc-sw@doc.ic.ac.uk
http://www.lesc.ic.ac.uk/

Abstract. The ICENI middleware utilises information captured within a component based application in order to facilitate Grid-based scheduling. We describe a system of application related meta-data that features a separation of concerns between meaning, behaviour and implementation, which allows for both communication and implementation selection at run-time, while providing the user with a flow-based programming model. It is shown that this separation enables a flexible approach to scheduling, and eases the integration of components with disparate control flow patterns or data types, by means of converters and tees for collective communication. By explicitly recording application information and supporting multiple scheduling approaches, communication protocols and component applications, while retaining OGSA compatibility, the ICENI component model is ideally suited to Grid computing.

1 Introduction

ICENI, the Imperial College e-Science Networked Infrastructure, is an experimental framework for Grid computing that supports the complete top-down utilisation of grid resources for scientific applications. The application framework within ICENI features the use of component based technology to capture information regarding an application's structure.

In this paper we describe the incremental development of this system, and a second generation component description language which features a separation of component meaning, behaviour and implementation. This separation isolates meaning, based upon typed dataflow between components, from the associating flow of control. User construction of an application relies exclusively upon the information in the meaning level. The behaviour and implementation information are used to build performance models to facilitate scheduling and implementation, and to inform communication selection.

2 Meta-data

In previous works [1,2], we have made the observation that information is key to the successful exploitation of the Grid - information about the resources themselves, information regarding the user's requirements, and information about the applications that are to operate upon the Grid. The ICENI component framework captures information relating to the application, its structure and the inter-component data and control flow.

2.1 Separation: Meaning, Behaviour and Implementation

We identify three concerns within the grid-oriented component model, all of which are independent, and through their interaction suffice to define the nature of the application. Each of these three categories of meta-data has a role to play in the grid deployment of a component.

Meaning From the user's perspective it is essential that a software component is endowed with meaning, in particular its composability with other components. As components are defined principally through their interactions with each other we attach meaning to the flow of data between components. Thus the highest level of abstraction is that of typed dataflow, shorn of control flow information.

Behaviour Separate from the component's meaning in terms of dataflow is *how* the data is passed from one component to another, and what dependencies exist between the dataflow relations described in the component's meaning. Thus the component's *behaviour* is a distinct concern, but necessary for scheduling and other grid-oriented tasks.

Implementation The implementation of the component also possess meta-data, in the form of the concrete format of data passed through the ports, performance characteristics of the computation that occurs when control is passed to the component, and the platform and operating system that this particular implementation may be deployed upon.

These three concerns are independent, though overlapping. Every component instance will have a single meaning, behaviour and implementation. On the other hand, a single meaning may have multiple behaviours, and a single behaviour may have multiple implementations. While the user manipulates components in terms of their meaning, behaviours and implementations may be selected by the Grid middleware, providing a flexibility necessary for effective application deployment on the Grid.

2.2 Meta-data Representation

A component is described by a set of documents that capture its meaning, behaviour and implementation. These documents use three different XML realisations, a Component Definition Language (CDL), Behaviour Definition Language (BDL) and Implementation Definition Language (IDL) for meaning, control concerns and software issues respectively.

Each component possess a set of ports, through which all communication passes. The meaning document captures the abstract dataflow through the ports, the behaviour document records the control flow through the same ports, while the implementation description document specifies the actual data format of passed messages, together with the performance characteristics associated with the port's behaviour.

Each XML document provides annotations for the same port, which has a unique name (within the component definition). The port characteristics are as follows:

CDL A port represents the production or consumption of data. As such at the meaning level a port has an associated dataflow, **in**, **out**, or **exchange**. An inport represents the consumption of data, an outport its production, while exchange represents a port which performs both. Inports and outports possess an abstract data type, which identify the type consumed or produced, respectively. Exchanges possess two types, indicating the flow in and out of the port.

BDL The BDL document provides additional control information for the ports - each port must correspond to a CDL port. Control flow is specified as being **in** or **out**. Any of the dataflow directions may possess either of the control-flow directions. The various combinations may be interpreted as different forms of message passing behaviour as indicated in Table 1. Those components which possess control flow in also possess a **dependency**, which indicates which other component ports are accessed following the arrival of control flow.

IDL The IDL defines concrete data types, including the precise format of the data, for all of the component's ports. Additionally the IDL includes performance characteristics for those ports which have a control flow **in**.

These markup languages are designed to be extensible. In the future we may extend the CDL with additional "meaning" information, regarding a component's mathematical properities for example, or augment the IDL performance metadata with information regarding the implementation's security characteristics, or fault tolerance, for example.

Table 1. Interpretations of the Data and Control Flow Annotations

	Dataflow In	Dataflow Out	Exchange
Control Flow In	Message Receive *push*	Method Offered (no arguments) *pull*	Method Offered (arguments) *pull*
Control Flow Out	Method Call (no arguments) *pull*	Message Send *push*	Method Call (arguments) *push*

2.3 Example: Finite Difference Method

The Finite Difference Method solves sets of partial differential equations in n dimensions over a given region of space where the final state of the boundary is known. Figure 1 illustrates the construction of the Finite Difference Method solver as a component architecture within ICENI. The solver component requires two inputs, the stencil operation (derived from the partial differential equations) and a description of the space that the differential operator is to be applied to. The latter includes such information as the portion of space that is to be solved, the boundary conditions for the space and the initial grid resolution. A display component is also attached to display the final result.

Example CDL, BDL and IDL documents for the FDM solver component are shown in Table 2.

2.4 User Level Application Composition

An application is composed by the user from component instances by using information at the 'meaning' meta-data level. The end-user making the composition ideally avoids all reference to corresponding behaviour and implementation annotations for their component types. An application consists of a set of component instances, together with a set of links, defined as an ordered pair of component ports. The types of the component ports (in terms of their abstract meaning-level type) must be the same, and the dataflow directions must be compatible, i.e. a dataflow in port must be connected to a dataflow out port, while an exchange must be connected to an exchange.

The links thus represent channels of data flow passing between concurrently existing components. At this level of abstraction the component composition is an example of *Flow Based Programming* [3]. This means that all control flow issues are hidden from the end-user, simplifying the task of connecting components with different control flow patterns. As long as the data types and direction of information flow is correct, components can be composed. Each link connects only two ports, and each port may only have one attached link. Collective communication between multiple ports is discussed in Section 3.1.

The details of binding different components together, in terms of behaviour and their software, is left to the middleware (see Section 2.7). This delegation to the middleware is made possible by the encapsulation of the control issues, and the isolation of the meaning (in terms of dataflow) as relevant to the user.

2.5 Application Description Document

Application composition produces an application description document, which is passed to the scheduling system. This document consists of specifications to create new component instances, and to establish links between component instances, either those newly created or currently executing. Though application description document is an XML document, we represent it as a flow diagram, as it is intended that composition will take place using visual programming

Table 2. Meta-Data and Derived Java Interfaces for the FDM Solver

CDL: Meaning

```
<componentDefinitionDocument
 name="FDM Meanings">

<componentTypeDefinition>

<componentTypeName>
  FDMSolver
</componentTypeName>

 <port>
  <name>stencil</name>
  <portTypeDefinition>
   <ddl:adt>
    StencilOperation
   </ddl:adt>
  </portTypeDefinition>
  <dataflow>in</dataflow>
 </port>

 <port>
  <name>spaceDescriptionIn
   </name>
  <portTypeDefinition>
   <ddl:adt>
    SpaceDescription
   </ddl:adt>
  </portTypeDefinition>
  <dataflow>in</dataflow>
 </port>

 <port>
  <name>matrixOut</name>
  <portTypeDefinition>
   <ddl:adt>
    matrix
   </ddl:adt>
  </portTypeDefinition>
  <dataflow>out</dataflow>
 </port>

</componentTypeDefinition>

<componentTypeDefinition>

... other components ...

</componentDefinitionDocument>
```

BDL: Behaviour

```
<behaviourDescriptionDocument>

<behaviour
ComponentDescriptionDocument
  ="FDM_cdl.xml">

 <componentName>
  FDMSolver
 </componentName>

 <behaviourName>
  Pull Model
 </behaviourName>

 <software>idl.xml</software>

<portBehaviour>
 <name>
  stencil
 </name>
 <controlFlowOut/>
</portBehaviour>

<portBehaviour>
 <name>
  spaceDescriptionIn
 </name>
 <controlFlowOut/>
</portBehaviour>

<portBehaviour>
 <name>
  matrixOut
 </name>
 <controlFlowIn>
  <dependency>
   <sequential>
    <call portName="stencil"/>
    <call
portName="spaceDescriptionIn">
   </sequential>
  </dependency>
 </controlFlowIn>
</portBehaviour>

</behaviour>

</behaviourDescriptionDocument>
```

IDL: Implementation

```
<implementationDescriptionDocument
 name="FDM implementations">

<implementation
 ComponentDescriptionDocument
   ="FDM_cdl.xml">

 <componentName>
   FDMSolver
 </componentName>

 <implementationName>
  DefaultImplementation
 </implementationName>

<portImplementation>
 <name>stencil</name>
 <dataStruct>
  icpc.FDM.Stencil
 </dataStruct>
</portImplementation>

<portImplementation>
 <name>spaceDescriptionIn</name>
 <dataStruct>
   icpc.FDM.SpaceDescription
 </dataStruct>
</portImplementation>

<portImplementation>
 <name>matrixOut</name>
 <dataStruct>
  icpc.matrix.DgeRCj
 </dataStruct>
</portImplementation>

</implementation>

 ... other components ...

</implementationDescriptionDocument>
```

Required Interface (Component)

```
public interface FDM_Interface
                       extends GridComponent {

    public icpc.matrix.DgeRCj matrixOut();
}
```

Middleware Interface (Context Object)

```
public interface FDM_Middleware
                       extends ContextObject {

    public icpc.FDM.stencil stencil();

    public icpc.FDM.spaceDescriptor
                         spaceDescriptionIn();
}
```

tools. Indeed, the current ICENI version includes a functional visual composition tool [1].

Figure 1 below shows the application description document for the Finite Difference Method application. In this, and other figures, each application description document is represented by a dotted box surrounding component instances

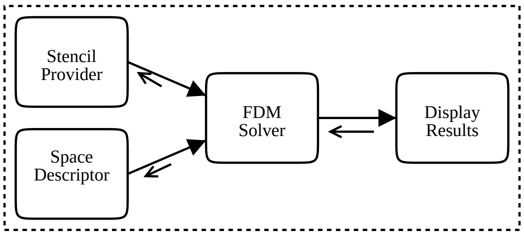


Fig. 1. The Finite Difference Method Applicaton Structure

and links. Documents can refer to existing components in other documents by a unique identifier. Links are represented by arrows with solid heads, indicating the direction of dataflow. Control flow is also indicated with additional annotation arrows with hollow heads. Control flow directions would not be visible to the user, but are shown here for illustrative purposes.

2.6 Software Bindings

The XML annotations that describe the component are used to construct the software bindings that allow the component to interact with the middleware and hence the grid environment. The XML may be mapped to more than one set of bindings - our current system can create both Java interfaces and WSDL (Web Service Definition Language) documents for a given component definition. This extensibility is a key strength of the annotation technique. If one were to define the component with Java interfaces directly, OGSA compatibility via web services would be lost, while restricting to the WSDL document forces one into a particular paradigm. (See Section 4 for further discussion on this point).

Where Java software bindings are created, two interfaces are produced for each component definition. These automatically generated interfaces form a contract for the component developer. The first Java interface, relating to the software component, must be implemented by the component developer. The second Java interface defines a context object which is provided by the middleware at run-time. In order to access the middleware functions or communicate with other components, the user code may call the provided methods of the context object. Thus there is a fair trade for the component developer - in exchange for implementing an interface (which grants access to the middleware and hence other components) the user code is provided with the means to itself access the middleware and other components.

A similar situation results with the automatically generated GSDL (Grid Services Definition Language) documents, in that they define a component as a GridService, and as such they may utilise the OGSA tools to access and be accessed by other OGSA compliant entities.

2.7 Behaviour and Implementation Selection

Once the application description document is submitted for execution, it is necessary to match the available BDL and IDL information for the specified component types. Typically a behaviour will have many implementations (the component code compiled for different architecture options), while an implementation will have only one corresponding behaviour. The middleware must choose between implementations, each with their associated behaviour.

The choice of implementation is made by analysing the control flow paths between the the components, as discussed in our previous works on the subject [2,4]. With the enhanced XML, the dependency information used to build the call graph of the application is not stored in “outports”, but is attached to any port with control flow **in**. Thus while pull mode methods have dependency where data flows out, push mode messages have dependency attached when they arrive.

Where legacy codes are used, they will possess only a single behaviour and implementation, and in as such there is no selection at this stage. Nevertheless the annotations provided in the CDL, BDL and IDL are used during scheduling (see Section 4).

2.8 Communication Selection

From the component developer’s point of view, implementing communication features ends with the satisfaction of the automatically generated software bindings. The actual connections between the run-time components are decided and instantiated by the middleware according to the relative positions of the endpoints. For example, where two components are scheduled to execute on the same machine, the middleware can use conventional procedure calls, or low-latency MPI [5] libraries. For distributed execution remote procedure calls, Java RMI or SOAP [6] may be employed. This system’s strength is its flexibility. The scheduler can schedule according to the available resources and requirements, and as long as the software bindings are adhered to, any convenient protocol may be used.

3 Advanced Issues and Case Studies

While the separation of concerns and static component model outlined above prove extremely flexible in terms of middleware selection and application construction, a number of applications require more sophisticated structures. These aspects are the subject of ongoing research within the London e-Science Centre. The research uses the ICENI component system to support practical applications. These serve as case studies that illustrate various features of the model, and deal with ongoing areas of our research.

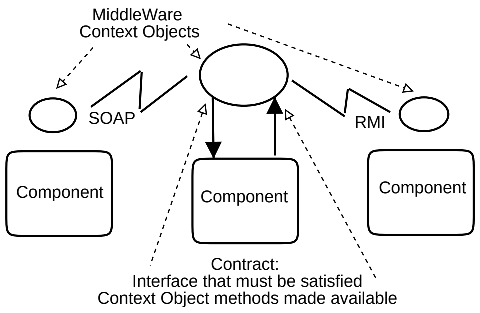

MiddleWare
Context Objects
SOAP
RMI
Component
Component
Component
Contract:
Interface that must be satisfied
Context Object methods made available

Fig. 2. Communication Selection via Middleware

3.1 Collective Communication

While each link only connects two ports, and each port may only have one attached link, collective communication between multiple ports is facilitated by means of tees. Examples of some possible tees are given in Figure 3, though this list is by no means exhaustive.

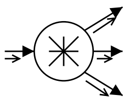
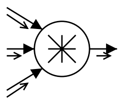
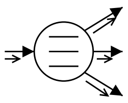
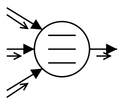
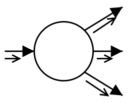
	A **switch** has a single inport, and multiple outports. The inport takes in a pair consisting of data type and an integer, which specifies the outport to which the data is sent.
	A **combiner** reverses the switch process, with multiple identically typed inports, and a single outport. A buffer is used to store incoming data, so that multiple links can feed into a single inport of the same type.
	The **splitter** has multiple outports, and a single inport which takes a data-type which itself must be an array (in our instantiation, an `<xsd>` sequence), with the same number of elements as outports. Thus the splitter scatters the data to multiple components.
	The **gather** is the reverse of the splitter, with multiple inports that bring in data which is combined to form an array, and passed to the outport. Like the combiner it buffers already received information.
	A **broadcast** possesses multiple outports and a single inport carrying the same data type. The incoming data is buffered, and made available through the outports (whether by being replicated and sent as messages in a push mode, or being made available to pull mode methods).

Fig. 3. Tees for Collective Communication

These tees are created using automatic code generation, which takes in the specified data type, together with a given number of ports, and produces the tee code together with the associated component description XML. From the user perspective control flow remains concealed - it is generated automatically along with the code for the tee. Hence while the user selects the tee manually to satisfy their requirements at the level of meaning, the behaviour and implementation of the tee are middleware generated.

3.2 Tees Case Study: GENIE

Grid Enabled Integrated Earth system model (GENIE) [1] aims to simulate the long term evolution of the Earth's climate, by coupling together individual models of the climate system. The constituents may include models for the Earth's atmosphere, ocean, sea-ice, marine sediments, land surface, vegetation and soil, hydrology, ice sheets and the biogeochemical cycling within and between components. GENIE aims to be a modular and scalable simulation of the Earth's climate, allowing for individual models in the system to be easily added or replaced by alternatives.

Figure 4 illustrates the organisation of the application. The simulation components communicate with each other through an integration component. This component also performs tasks requiring data from all the simulation components (e.g. describing the heat exchange between the surface of the ocean and the base of the atmosphere), and is designed to be extendable to allow further simulation components to be added to the system in future.

A control component manages the flow of information between each of the components in the GENIE framework. It also allows communication of the simulation data with external resources such as visualisation and steering components (see Section 3.3).

This particular case study demonstrates the flexibility of the component system at design time. Multiple simulation components may be attached to the integration component by the application builder as she sees fit. This requires no change to the integration component itself: as long as it can handle multiple simulation components (with a parameter passed as data by the setup component), the actual collective communication is organised using tees.

3.3 Factories

While the composition and deployment system outlined above is essentially static, in that a complete application is composed and deployed as a single unchanging unit, it may be extended to realise dynamic programming by making multiple submissions of connected applications to the scheduling system.

A *factory* component is a component capable of creating an XML application description document and submitting it to the middleware's scheduling

[1] a recently funded Natural Environment Research Council e-Science pilot project

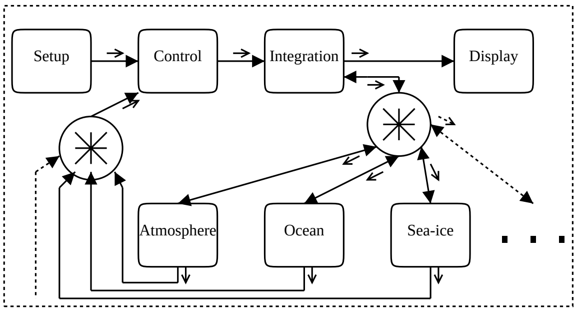


Fig. 4. GENIE Application Structure

system. This is done at run-time, and thus the created application may be data dependent. An example of this form of behaviour is given in Section 3.4.

The links in an application description document may refer to either a port on a new component instance (specified within the application description document), or to any already existing component instance. Thus the factory can create a new subset of components at run-time, and connect them to the already existing components (such as itself), given that it can access the middleware to identify them.

As a port can only possess one attached link, the scheduling of a new link replaces the pre-existing one. Thus factories may rewire the network of existing components as well as generating new components. In this way completely dynamic behaviour is expressible, while at the same time restricting all component creation to occur through the middleware scheduling system.

A factory component accesses the middleware through methods made available through the context object (for Java software bindings), or by being declared as a Factory (for the OGSA-WSDL bindings).

3.4 Factories Case Study: Parameter Sweep of Acoustic Scattering Application

This application is a parameter study, in which the the acoustical back scatter from a number of different submarines is computed across a range of designs, and the optimal design is subsequently identified according to some user defined criteria. The acoustical simulation is performed with an independently written application, DRACS [7], while other components extend the single application to a full parameter study. A Design Generator component produces design specifications for a number of submarines. Each submarine design is then converted to a three dimensional unstructured mesh (the required input to DRACS) by a Mesh Generation component. DRACS then runs inside a component wrapper to perform the analysis and the back scatter data is passed to an analysis component, which may request that the Design Generator produces a new generation of submarines if none of the results are acceptable, within the user's tolerance range. This is shown in Figure 5.

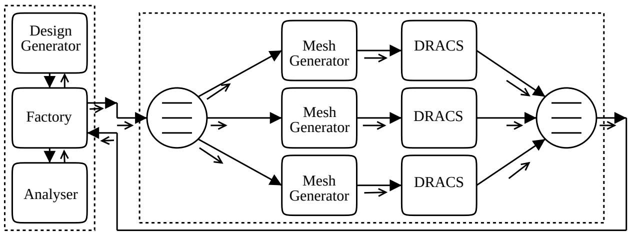


Fig. 5. Acoustic Scattering Parameter Study

This application highlights dynamic component creation, as the number of submarines analysed per generation is data dependent. As such, an analysis factory component launches a secondary application (containing the parameter sweeps) and connects to it during the execution cycle of the primary application.

4 Related and Further Work

The ICENI component model is complimentary to the Open Services Grid Architecture [8], which is rapidly becoming the accepted standard for gird based development. The ICENI model deliberately adopts the notion of standard grid life cycle interfaces within its software bindings and the notion of factory components, which may be realised by the OGSA FactoryServices. Thus an executing component may be exposed as a service within the OGSA model. Component software services may be discovered and utilised using a Service Oriented Architecture lookup process. We have described how ICENI components may be exposed as both Jini and OCSA services in a companion paper [9]. What the ICENI model adds to the Service Oriented Architecture design is information, beyond the software bindings provided by OGSA and the underlying Web Services technology, which remains immature in terms of control flow descriptions.

The explicit meta-data provided by the ICENI system allows implementation and communication selection as described above. Additionally it enables the constuction of dataflow and control flow graphs that facilitate scheduling on the distributed resources made available by ICENI [1].

5 Conclusion

The ICENI component model has been extended to include a separation of concerns between meaning and behaviour, as well as implementation. This gives the ICENI grid user the following added value:

– Flow based programming model hides control and thread issues from the end-user, easing application development. Tees explicitely enable collective communication within this model.

– Enables easy generation of a range of ‘contract’ software bindings - the XML annotations do not force a particular model on the component developer.
– Provides information that facilitates dataflow scheduling, while retaining the control- and dependency-based performance modelling of previous ICENI work [4].
– The model is OGSA compatible, but not restricted to the OGSA view.

In effect the component model adds value by adding information to existing or novel applications. This information, in terms of meaning and behaviour, provides a greater handle on the code and its composition than a standard ‘interface’ or simple software binding can produce. It allows a range of automated composition and selection techniques at various levels of abstraction - costing the user or component developer nothing, but enabling exploitation of the grid resources.

Acknowledgements

This work is funded by the UK DTI/EPSRC e-Science Core Programme (THBB/C/008/00023)andbyEPSRC grants GR/R74505/01andGR/R67699/01. The equipment used in this work was funded through HEFCE JREI awards GR/R04034/01 and GR/M92355/01.

References

1. N. Furmento, A. Mayer, S. McGough, S. Newhouse, and J. Darlington. A Component Framework for HPC Applications. In *Euro-Par 2001, Parallel Processing*, volume 2150 of *LNCS*, pages 540–548. Springer-Verlag, 2001.
2. A. Mayer. *Composite Construction of High Performance Scientific Applicaitons.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 2001.
3. J. Paul Morrison. *Flow-Based Programming : A New Approach to Application Development.* Van Nostrand Reinhold, July 1994.
4. Nathalie Furmento, Anthony Mayer, Steven McGough, Steven Newhouse, Tony Field, and John Darlington. Optimisation of Component-based Applications within a Grid Environment. In *SuperComputing 2001*, Denver, USA, November 2001.
5. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.
6. W3C Consortium, May 2000. `http://www.w3.org/TR/2000/NOTE-SOAP20000508`.
7. Steven Newhouse. *Adaptive error analysis with hierarchical shape functions for three dimensional rigid acoustic scattering.* PhD thesis, Imperial College of Science, Technology and Medicine, April 1995.
8. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. `http://www.globus.org/research/papers/ogsa.pdf`.
9. N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington. ICENI: An Open Grid Service Architecture Implemented with Jini. accepted for SuperComputing 2002, Baltimore, November 2002.