Grid workflow captures “programming the Web or Grid” and encompasses a broad range of approaches with names like “Service Orchestration”, “Service or Process Coordination”, “Service Conversation”, “Web or Grid Scripting”, “Application Integration”, or “Software Bus”. It is an area of active research with different approaches emphasizing control flow, scheduling and/or dataflow. The workshop at GGF10 is a good summary of current best practice [workflow] with a comprehensive recent review in [Yu05A]. There is some consensus that although its use for dataflow is problematical, one should build upon BPEL [BPEL4WS] [activeBPEL] – the industry supported Web Service standard for which open source implementations are being developed [OMII]. This was based on [WSFL]. Important dataflow technologies are the Kepler [Kepler] and Triana [Triana-A] projects while Pegasus and Chimera from the Globus-Condor collaboration address the scheduling style of workflow [Pegasus] [Chimera]. Taverna from the myGrid project successfully integrates control and workflow [myGrid-B]. The HPSearch project from Indiana University supports workflow efficiently from a familiar scripting environment [HPSearch] [Gadgil04A]. An important issue for Grid integration with HLA will be melding the workflow ideas from the Grid with the federation ideas from HLA. Workflow technologies have wide applicability stretching from RTI-like systems, application code coupling and support of problem solving environments.

1. Grids of Grids

In [GofG], we introduced the concept of Grids of Grids of Simple Services”. Here we review and extend this and show how it can address several issues of importance to DoD including the “right sizing of services”, “an architecture for dealing with legacy systems” and a “strategy for modularizing the design and implementation of systems (grids)”.

Consider any (software) problem you like and imagine how it would look in a traditional approach of a decade or so ago. One would get monolithic chunks of software in some language like C++ or ADA. This would be divided into methods or subroutines and we would be instructed to build it in modular fashions using libraries and well defined interfaces. As technologies developed we added new languages like Java and better software engineering processes which still however focused on modularity within a chunk of software divided into objects and/or components. For example you will find this software structure if you inspect well known open source projects such as Linux or the Java at the Apache site [Apache] . One can convert such code into services by specifying each of interfaces in XML and providing a Web Service wrapper. This activity is important for jump starting our collection of services but I would view it as only appropriate for legacy systems and a less than optimal approach to new software systems. For example looking at the many different Apache projects, one will find many related but different implementations of common subservices like security, file access and user profile. Building a system combining several projects would often require an integrated approach to common services like security. This would be relatively easy if the implementation of each subservice like security was a separate Grid service with well defined message-based interfaces. However with traditional approach, the typical subservice can have an external message-based interface but unfortunately in addition many internal method linkages to other parts of the software chunk where typically it is hard to serialize the arguments. Thus subservices like security cannot be extracted from the glob and it is very hard to use components such traditional software systems even if they run excellently with service interfaces.

The above discussion allows us to identify a strategy for defining what we term simple services. Start by examining the different capabilities of one’s systems. Services are distributed components that have distinct functionality – especially functionality that is usefully shared among different uses. Services must be able to achieve acceptable performance when implemented with message based interfaces and distributed platforms. There is an inevitable difference in overhead between message and method based interactions; messages could experience 100’s of milliseconds in network latency while the internal method calls have a fraction of a millisecond overhead. We define simple services as those that are as small as possible given the performance implications from the decomposition. Such simple services are then the unit for which one uses traditional programming models and languages. This is the proposed strategy for “right-sizing” services.

In fig. 5, we suggest a packaging and coupling approach that generalizes and distributes that familiar from the traditional software hierarchy:
lines of code 🡪 methods (subroutines) 🡪 objects (programs) 🡪 packages (libraries).
A single simple service is the smallest grid but we can integrate like simple services into library grids. These sub-grids are then composed into a complete “Grid of Grids” implementing the full system. Fig. 5 shows how database, sensors and compute nodes (abstracted as simple services representing “simple resources”) can be federated, networked and clustered into larger units.

As a particular example of a Grid of Grids, Fig. 6 illustrates how one can share component Grids between critical infrastructure applications and DoD’s NCOW [Fox05D]. The Department of Homeland Security has identified critical infrastructures that include Agriculture and Food, Water, Health, Industrial and Defense Base, Telecommunications, Energy, Transportation, Banking and Finance, Chemical Industry and Hazardous Materials, Postal and Shipping. The critical atomic Grids in this case include those for sensors, GIS, visualization, computing and collaboration. We also need of course the core Grid shown at the bottom of the figure with services like security, notification and meta-data. These atomic Grids can be re-used as shown in figure 6 in all critical infrastructure Grids and illustrate the important interoperability principles with which Grids are built. These CI(Critical Infrastructure) Grids are in turn customized, composed and overlaid with other Grids (such as weather, census data) for different CI communities. This way one generates Grids aimed at Public Health, Emergency Response (Command and Control) or Crisis Grids, Infrastructure Planning, Education (schools) and Training (of managers and first responders). Clearly the Grid of Grids concept can be applied recursively and dynamically.

Note that both simple services and grids interact with the outside environment through messages and these messages are the only way to both impact and learn about the service or grid. For Web service based grids these messages are defined by the WSDL – where for grids this WSDL is the concatenation of the WSDL of all its external interfaces. Note that in this view all that counts are “outward facing” interfaces. Internal interfaces need not be specified to use a given grid. In particular these internal interfaces could use different flavors of Web service specification or totally different technology – methods, Java RMI, CORBA etc. Examples of different Web service flavors are WSRF or WS-I+ based systems [WSRF] [WSGrids] or more simply the two flavors of reliable messaging in [WS-Reliability] and [WS-RM]. If we assume that we use a message-oriented-middleware (MOM) implementation then all messages entering a particular simple service or grid is explicitly handled and can be transformed to confirm to the internal conventions of this grid as illustrated in fig. 7. This gives us a clear strategy for legacy systems – one identifies their outward facing Grid interfaces, defines in WSDL and builds a set of transformations that map between the system-wide Grid standards and those used internally. The same idea can be used to build virtual private grids generalizing VPN’s to grid systems [Fox04B] and so ensuring particular security policies within a given subgrid. More generally support of hierarchically constructed grids of heterogeneous components gives a robust software engineering strategy with a modular software model.