Patterns and Operators for Grid Software Development

Omer F. Rana\textsuperscript{1}, Maria Cecília Gomes\textsuperscript{2}, José C. Cunha\textsuperscript{2}
(1) o.f.rana@cs.cf.ac.uk, School of Computer Science and, Welsh eScience Centre, Cardiff University
5 the parade, po box 916, Cardiff CF24 3XF, UK
(2) \{mcg,jcc\}@di.fc.unl.pt, Dep. Informática,
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa,
Campus FCT, 2829-516 Caparica, Portugal

1 Introduction and Motivation

As the hardware (high speed networks, workstation clusters, intelligent sensors etc) and software (Globus/OGSA, UNICORE etc) infrastructure for Computational Grids matures, the next challenge is to identify a way to connect these disparate collection of resources and services to create the next generation of scientific applications. Every scientific discipline will have some common themes which describe how a particular set of experiments are to be set-up and managed. Often, such deductions are a combination of experience and knowledge about a particular domain. Such “idioms” are however often hidden by the complexity of actually managing the experiment – such as setting up scientific instruments, configuring data sources and various computing environments. There is therefore clearly a need to capture these common idioms, as they often encapsulate best-practice that should be shared within a particular community. It may also be interesting to identify common themes across multiple scientific disciplines, thereby providing a means to share software tools and data sets more effectively. Due to the integration capabilities offered by the Grid, identifying, recording and publishing such common idioms may provide a useful way to undertake better science.

The approach adopted in this work is primarily aimed at providing suitable tools for computational scientists and developers, who have some understanding of the computational needs of their application domain, to express common usage of software infrastructure within their domains as “design patterns”. To use these patterns, it is assumed that a scientist is aware about the likely coordination and interactions taking place between software components of the application (such as a database or numeric solver, for instance) that is being developed. Software engineering tools for Grids must support application configuration, execution control and reconfiguration of software services. Our
contribution comprises a software engineering tool where the composition of Grid resources results from manipulating patterns through pre-defined pattern operators. The tool provides a library of pre-defined and user defined pattern templates, and a library of pre-defined operators. The operators also allow the subsequent execution control and reconfiguration of the application. Patterns must be defined using the Unified Modeling Language (UML). This work is aimed at complementing existing work in workflow systems and Problem Solving Environments.

Patterns abstract common interactions between components in Grid systems, enabling reuse — thereby enabling application scientists to define common idioms relating to software use within a particular domain. Hence, a user may build an application in a structured fashion by selecting the most appropriate set of patterns, and by combining them according to operator semantics. Furthermore, we divide patterns into ‘structural’ and ‘behavioural’ categories, allowing users to choose the most appropriate combinations. Structural patterns encode component connectivity (e.g. a set of components connected in a ring fashion), whereas behavioural patterns capture temporal or flow dependencies (data and control flow dependencies) between components (e.g. a Client/Server model or a Producer/Consumer model). Operators define actions that can be applied over patterns and they are also divided into structural and behavioural categories. Structural operators manipulate structural patterns (e.g. adding an element to a ring pattern) and behavioural operators manage temporal or flow dependencies (e.g. to stop and resume the execution of a behavioural pattern). Behavioural operators allow the user to control application execution and re-configuration. Pattern operators may be applied in an ordered combination and the sequence may be shared among users. To implement pattern templates and pattern operators we have extended the Triana [2] workflow tool to support patterns. At present, only structural patterns and operators are implemented. The Grid community has also recognised the importance of patterns [1] as a way to re-use expert knowledge. Patterns are not just a modeling abstraction, but have also been included into development tools, as first class entities. Our approach also treats patterns as first class entities, but differs in that the user may explicitly define structural constraints between components, separately from the behavioural constraints. Such structural constraints may be useful, for example, to represent common software architectures. The major benefit of our approach is an easy manipulation of pattern instances through operators, which simplifies the design process. Additionally, pattern refinement can be undertaken as a sequence of operators. Our approach aims at providing a novel way to access and compose Grid services.

2 The Need for Grid Patterns

Identifying the granularity at which Grid Patterns should be defined is also an important design choice. Due to the hierarchical nature of many existing Grid systems, it is likely for such patterns to exist at multiple levels. An initial
distinction could be made between “Co-ordination” patterns and “Structural” patterns.

- **Co-ordination (Behavioural) Patterns:** These patterns capture interactions between software sub-systems and may be application specific – such as a “Graphics Pipeline” Pattern, or application independent – such as a “Broker Service” Pattern or the “Service Aggregator/ Decomposer” Pattern. Such co-ordination patterns capture interactions between software sub-systems, enabling different implementations of such sub-systems to co-exist. This is particularly relevant in Grid computing where multiple instances of the same sub-system are likely to co-exist.

- **Structural Patterns:** These patterns capture connectivity between particular types of Grid software/hardware components. Such patterns may encode physical connectivity between particular sites (such as network topologies), or may encode logical connectivity that is likely to persist over longer time periods. Examples of such connectivity include a “ring” topology for connecting computational resources, for instance.

Behavioural patterns in our approach are intended to be dynamic, and must also specify the direction of dataflow. Both of these types of Patterns may be subsequently manipulated through a set of operators. We distinguish between “Behavioural Operators” and “Structural Operators”. Behavioural operators are intended to modify the dataflows between structural components via scripting tools. Structural operators are aimed at modifying the number of elements, or their types, within a given topology. Additional details about the different types of operators supported can be found in [3].

3 Implementation Status

A prototype of the Pattern based environment has been implemented over the Triana workflow engine. The prototype allows developers to utilise a collection of pre-defined patterns from a library. The patterns appear as standard components that can be combined with other patterns or executable units using an editor – as illustrated in figure 1. Each structural pattern contains a collection of “dummy” components – essentially place-holders that can be instantiated with executables. Behavioural patterns on the other hand are implemented over the run-time system used to execute the components. There is no visual representation of these, as they are provided as a collection of scripts that need to be configured by a user prior to execution. A behavioural pattern encodes a particular execution sequence and is constrained by the resource managers available. Hence, the behavioural pattern Stop is only relevant if a resource manager allows the current execution to be paused or terminated.

Similar to behavioural patterns, there is no visual representation of either structural or behavioural operators. These are intended to be used as program scripts for modifying the structure of the application being executed. They
operate on particular pattern instances, and may be used to modify data flows between existing applications. Structural and behavioural operators are mapped to the DRMAA API [4] – as resource management systems widely deployed for scheduling Grid applications (such as LSF and Grid Engine) also make use of this. For instance, many commands in the Sun Grid Engine related to job execution have \texttt{drmaa} prefixes [5]. DRMAA uses an IDL-like definition (with IN, OUT and INOUT parameters) for specifying the API, and also provide support for handling errors (via error codes). With significant interest in the DRMAA API from the Grid community, we have chosen to use this as target for mapping our operators. However, it is useful to note that not all of the attributes specified in DRMAA are supported by resource management systems – such as the variety of error codes that DRMAA is expecting the resource management system to return.

Behavioural operators can include: \texttt{Start}, \texttt{Stop}, \texttt{Resume}, \texttt{Restart}, \texttt{Limit}, \texttt{Repeat} etc (see [3] for more details about these operators). The \texttt{Start}, \texttt{Stop}, \texttt{Resume} and \texttt{Restart} operators simply map to the DRMAA \texttt{drmaa_control()} routine. The \texttt{action} operation within this routine can support any one of the 4 operators specified above. Hence,

\begin{verbatim}
drmaa_control(job_id, DRMAA_CONTROL_SUSPEND, drmaa_context_error_buf)
\end{verbatim}

may be used to support the \texttt{Stop} operation. The significant difference is that whereas in DRMAA the \texttt{suspend} action is intended to apply to only one job, in our approach the \texttt{suspend} must be applied to all jobs within a particular pattern template – as our operators are intended for application over pattern instances and not individual jobs. It is also necessary to identify dependencies between components within a pattern instance, and captured in the behavioural pattern. These dependencies indicate the order in which the DRMAA routines need to be invoked on the currently executing jobs generated from the pattern.
Similarly, other routines within the DRMAA API can be used to manage pattern execution over a particular time period. Each of the routines in DRMAA needs to be generalised to the case of a pattern, rather than to a particular job. Behavioural operators can be combined into a script, and each element of the script is mapped to one or more DRMAA routines. For instance, consider the following script:

```c
main()
P1;P2;
section1{
    Rename(P1,P2);
    Replicate(P2,3);
    Owner(P2, "smoofr");
}

section2{
    Start(P2);
    Log(P2);
    Limit(30,P2);
}
section1; section2;
```

In this example, we first create a `main` block, which is used to combine multiple operations together. Two patterns are defined, as P1 and P2. Subsequent operations will only relate to these patterns. `section1` in the example renames an existing pattern, and replicates it 3 times. No name is assigned to this replicated pattern. The `Replicate` operator essentially generates three instances of pattern P2, and assigns an owner `smoofr` to this pattern. By assigning ownership of a pattern, all subsequent structural operators can only be applied to the pattern by this user. In `section2`, the execution of the pattern is started, and its output logged. As mentioned previously, the `Log` operator will only be valid if the underlying resource management system supports this functionality. If not, this operator is ignored. The `Limit` operator executes pattern P2 for 30 minutes and then stops.

References