A Pattern-based Software Engineering Tool for Grid Environments

Maria Cecília Gomes¹, José C. Cunha¹, Omer F. Rana²

¹Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Campus FCT, 2829-516 Caparica, Portugal,
²Department of Computer Science, Cardiff University, 5 the parade, po box 916, Cardiff cf24 3xf, UK

Abstract.

A pattern-based software engineering tool for constructing workflow based applications is described. The tool provides a novel way of composing applications executing over Grid resources. The tool provides a structured design approach, centered on the manipulation of patterns through pre-defined pattern operators. Patterns and operators are divided into two categories: structural and behavioural. Structural operators act upon structural patterns to build reusable architectures. Behavioural patterns define the data and control flow dependencies between components in the architecture. Components within a pattern are subsequently instantiated with particular executables, and behavioural operators allow subsequent execution control and reconfiguration of the application.

1 Introduction and Related Work

The Grid [1] provides an important paradigm for large scale distributed computing. A primary objective within such a paradigm is to provide co-ordinated sharing of geographically distributed services (which may include hardware, software and information resources) across multiple administrative domains. Hence, Grid environments may be heterogeneous and dynamic – thereby necessitating software design approaches and tools which can more effectively exploit such environments. Such tools 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 abstract common interactions between components in Grid environments, enabling reuse. 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 reconfiguration. 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 [9] workflow tool to support patterns. At present, only structural patterns and operators are implemented. They are the major focus of this paper.

There has been considerable research since GoF [2], on identifying relevant patterns for different application areas. The Grid community has also recognised the importance of patterns [4] 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. Furthermore, application re-usability and maintenance is improved if (design) patterns are still identifiable in the final code. Component paradigms also provide patterns as first class entities, where patterns may be defined, stored and reused independently of the components. Tools like the ones mentioned in [16, 17] provide a pattern-based approach for component composition – the ObjectAssembler visual development environment provides a catalogue of patterns for connecting JavaBean components. Similarly, the Pacosuite tool supports component composition through composition patterns which define component interactions.

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 in high-performance computing applications. 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. Major focus of other related efforts in the context of Grid computing is application centered, such as the DataGrid [19] and the MyGrid [18] projects. As a consequence, the resulting solutions are in some way restricted by the application area. The DataGrid project, for example, aims to provide scientific “collaboratories” through middleware supporting cross-application resources in the areas of High-Energy Physics (mainly), Biology and Earth observation. Our approach is more generic in scope, and more widely applicable. Our approach is primarily aimed at two kinds of users: (1) software developers interested in abstracting common interaction patterns within a workflow tool (independent of any application), and (2) computational scientists familiar with particular application-domain specific software libraries, and interactions between these. Pattern based programming may not be a useful approach for highly tuned applications on particular architectures (such as for ‘production’ codes), or when the user is unfamiliar with the types of interactions between different software components.

2 Pattern Templates and Pattern Operators

This section presents a set of structural patterns, followed by structural operators.
2.1 Structural Patterns

Structural patterns may be divided into two groups: topological and non-topological. Topological patterns represent structures that frequently occur in Grid systems. For illustration purposes, we introduce three basic structures represented by the pipeline, the ring, and the star patterns. The non-topological patterns correspond to a sub-set of state-of-the-art design patterns ([2]), selected because of their utility in Grid systems. We select the Adapter, the Facade, and the Proxy patterns.

The pipeline pattern represents a sequence of ordered stages, where one stage produces data to the next. For example, in a signal processing application the first stage may consist of a signal generator service producing data to a set of intermediate stages for filtering. Frequently, the last stage consists of a visualisation service for observing results. Formally, the pattern’s structure can be defined by three components, as represented in the left-hand side of figure 1. The structure is represented in UML notation [5] and was adapted from the Pipes and Filters pattern [3]. The first component (DataSource) produces data to a Connector component from which data is then consumed by a third component (DataSink). The ring pattern can be seen as an extension to the pipeline, as it also consists of a set of stages, but with the “last” stage connected to the “first”. The ring pattern can be represented similarly to the pipeline pattern: instead of the DataSource and DataSink components, they are replaced with a single Component with two links to the Connector. In Grid environments, the ring topological structure can be found in a number of applications, both in the context of application execution (such as for modelling interactions within a local area network) to logical topologies such as supporting an authentication chain when approving participants with multiple Certificate servers. Each server delegates an authentication request to the next domain, and the final server replies to the original client. This chain based mechanism can also be found in resolving the address/location of an executable file using a directory lookup service – such as Globus MDS [7].

![Figure 1: The Pipeline and Connector patterns.](image)

A star pattern consists of a nucleus which communicates through simple connectors to satellite components (a single connector links a single satellite to the nucleus). All services based on a centralised resource sharing model in Grid environments may be seen as matching the star structure.

The Adapter pattern allows communication between two elements when they do not have the same interface (see [2] for a formal definition). In Grids, the Adapter pattern may be used
to model wrappers for legacy codes (such as Fortran binaries). If the client is expecting a different interface from the one provided by the server, the adapter can act as a translator. This pattern is also particularly useful for providing a mapping between the interface of an existing code and a pre-defined component data model for Grids, such as CCA [8]. The Facade pattern (see [2]) may be used to restrict access to a set of sub-systems (such as an IP domain), through a common interface (the facade). The Proxy pattern (see [2]) allows the local presence of an entity’s surrogate which transparently supports access to the remote entity. Grid services, for example, are usually accessed through a proxy (or gatekeeper).

2.2 Structural Operators

Structural operators are used to modify and manipulate structural patterns.

![Figure 2](image2.jpg)

Figure 2: The Increase (left-side) and Extend (right-side) operators over a proxy pattern.

**Increase(P, n):** the number of elements in a pattern P is increased by n. Like any structural operator, the increase operator keeps the structure of the pattern invariant. The left-hand side of figure 2 shows the application of the increase operator to a proxy pattern for the creation of two proxy elements. Similarly, when using **Decrease(P, n),** ‘n’ elements are removed from the pattern.

**Extend(P, element):** an element is added to a pattern, and its structure is augmented. Figure 2 shows the Extend operator being applied to a proxy pattern. The proxy pattern is extended by adding a component to an existing proxy. This situation occurs in mobile agent/object systems, where the sequence of proxies is used for locating the agent/object (via a chain of message forwarders, for instance). Using the **Reduce(P, element)** operator, an element is removed from a pattern.

![Figure 3](image3.jpg)

Figure 3: Result of the application of the Embed operator to a pipeline pattern to include a star pattern into the first element of the pipeline.

**Embed(P1, P2):** pattern P1 is added into a higher-level pattern P2. The concept of hierarchy is supported here by enabling component place holders to contain other patterns. Figure 3 shows the embedding of a star pattern into the first element of a pipeline pattern. Similarly, **Extract(P1, P2)** causes a pattern P1 to be removed from a higher-level pattern.
A Pattern-based Software Engineering Tool for Grid Environments

P2. Rename(P1, P2) causes a structural pattern P1 to be transformed to pattern P2. Using the Replace(P1, P2) operator, pattern P1, as a single entity, is replaced within pattern P2. With Replicate(P, n), pattern “P” is replicated “n” times, and the resulting replicas are independent of each other. Group/Aggregate(P1,...,Pn) enables a group of “n” patterns to be seen as a single pattern, and behave as a single entity. We define an operator catalogue which may be used to modify structural patterns. The sequence of usage of operators is also significant.

3 Pattern Supported Workflow

Triana [9] is a workflow based tool (see figure 4) for integrating different components to build scientific applications. By default, Triana comes with a toolbox of pre-defined services/tools (called units) for signal processing, mathematical calculations, audio and image processing, etc. Triana also provides a wizard for the creation of new services, which are then added to the toolbox. Users compose applications: 1) by selecting units from toolboxes, and dragging and dropping them onto a scratchpad, and 2) by connecting (linking) components using the Triana interface. An application in Triana consists of a network of components connected into a TaskGraph – with connectivity between components checked based on data types associated with the input and output ports of components. Figure 4 shows a simple example with two components from the signal processing toolbox: a wave generator’s output node is connected to the input node of a grapher unit. Execution starts as soon as data arrives in a component’s input port(s), and a component is allowed to send results concurrently to several other components. Triana also has a grouping facility to enable a collection of components to be treated as a single group. This “group component” will have nodes (input and output ports) that allow connecting the internal components to outside ones. Finally, users may save the configurations they have defined and reuse them later on.

Once an application has been constructed using the Triana GUI, it must subsequently be executed on distributed computational resources. Figure 5 shows, in a simplified way, how execution is achieved using a collection of JXTA [13] based Triana peers [10, 11]. The peer supporting the Triana GUI acts as a co-ordinator for launching different parts of the TaskGraph on other peers; using input from the user, it decides which parts of the TaskGraph to run locally, and which on remote peers. Conceptually, Triana is a two-layered application
where the Triana GUI is de-coupled from the Triana Service (see figure 5) responsible for the (distributed) execution. Each peer must support a Triana Service, to enable execution requests to be received from the co-ordinator – essentially the Triana Service acts as a hosting environment to launch and manage task execution on remote resources. A peer may subsequently also sub-contract execution to other peers. A Triana-Service to Globus/OGSA bridge is also being implemented as part of the European GridLab project.

3.1 Patterns and Operators in Triana

Triana has been extended to support structural patterns and structural operators – which
are made available in the Triana toolbox. Intrinsically, a pattern in Triana is a group component, as it contains a set of entities (connected in a pre-defined way), with a particular set of parameters. Group components in Triana can also be recursive, i.e. they can contain other groups of units. Such a component has a set of input/output ports to support the connection to the input/output ports of the encapsulated units. For example, in figure 7 the Gaussian unit is connected to the Wave unit through an input node owned by the Pipeline. A Pattern Instance on the other hand, is a pattern with units instantiated to particular executables. Figure 6 shows the definition of a pattern instance in Triana, and figure 7 shows an example of a pattern instance. Initially a pattern consists of place holders for executables, identified as Dummy Units. A pattern is therefore not executable, but only becomes executable by instantiating dummy units to particular executable components (the result being a pattern instance). Hence, each pattern instance aggregates units according to a specific structure, and is managed by a Control Task.

3.2 A Few Triana Classes

Figure 8 shows the set of classes for implementing patterns in Triana, and is divided into three major areas: Unit definition presents the basic class for defining a Triana service/unit; Helper entities shows classes for realising the execution of a unit; and GUI entities gives an example of classes that support interaction with the users.

A new service is defined by extending the Unit abstract class, containing optional and mandatory methods to be implemented. The process() method is mandatory and specifies the specific actions to be performed by the unit. For an optional method, like init(), a user may define actions that must be executed prior to service execution (i.e. before process() is executed). Triana also provides an event mechanism, that allows event listeners to be associated with particular unit parameters. Helper classes like Tool, Task, and RunnableTask provide the necessary code to execute a unit, and to send and receive data. Tool defines code common for all units in the toolbox (e.g. parameter management code, like code to get the name of all the service’s parameters). Task extends Tool and represents a task in a TaskGraph, i.e. an entity that can be connected to other entities forming a workflow. RunnableTask makes the connection between a Unit and a Task. It initialises an associated unit (e.g. an object of class
Wave) by calling its `init()` method. Furthermore, it implements the data handling capability of a Task (e.g. keeps track of which nodes have data that has not yet been processed, and wakes up a task when there is data ready to be read on all input ports).

A TaskGraph is itself a group Task – an idea used to represent hierarchy. A TaskGraph contains a collection of tasks linked by cables. Whenever a task is created, it is always created in the context of a TaskGraph (and contains a reference to this taskgraph as shown in figure 8). A TaskGraph provides methods for creating a new task within it (`createTask()`) – connecting/disconnecting tasks which belong to the taskgraph; creating a new sub-taskgraph out of a group of tasks that belong to the taskgraph (`groupTasks()`) – etc. A TaskGraph may also act as a listener to the tasks that it represents (e.g. it gets notified when a task is disconnected from another task). Besides tasks, it is possible to listen to events from the tasks’ ports, and associated configuration parameters for these ports.

### 3.3 Implementation of Structural Patterns and Operators

Patterns and operators are implemented as extensions to the Unit class. For example, `DrawPatternTemplate` (figure 9) is an abstract class that extends Unit and implements structural patterns’ common behaviour. Particular structural patterns are then support by specific subtypes like `DrawPipeline`, `DrawRing`, etc. To create a pattern template like Star in figure 9, an instance of `DrawStar` creates a new TaskGraph that includes the instance itself. This instance also acts as a “control task” for the pattern. On execution, first it draws a default set of component place holders, i.e. `DummyUnit` tasks, inside the TaskGraph. A DummyUnit must subsequently be instantiated with a specific unit from the Triana toolbox. It is also the responsibility of the control task (i.e. the DrawStar instance) to support event handling for these DummyUnits. It also implements structural operators like `increase`, `embed`, etc. The operators’ major code is defined in `DrawPatternTemplate`, and specific actions are left to the subtypes. For example, the `increase` operator needs two actions: a) to create and draw a new DummyUnit element, and b) to identify the connection element, i.e. to which of the already existing DummyUnits should the new element be attached to. Action a) is common to all subtypes, so it is implemented in the `DrawPatternTemplate` class. Action b), in turn, is specific...
4 Conclusion

Ongoing work on implementing a pattern-based software engineering tool for Grid environments is outlined. The tool provides a library of useful pattern templates and pattern operators, primarily aimed at supporting workflow based applications. A variety of scientific applications in Astrophysics, Bio-Informatics, Chemistry fall into this category. Structural pattern templates are manipulated through structural operators that preserve the patterns’ structural constraints. A user is able to subsequently select and apply different behavioural patterns, defining and managing in this way various data and control flows between the components. After the instantiation of pattern elements with specific Grid services, a user will be able to control the application’s execution and reconfiguration through behavioural operators. The current implementation efforts towards achieving this is described, based on the Triana workflow tool.

Acknowledgements

We would like to thank the Triana group ([9]), in particular Matthew Shields and Ian Wang for their suggestions and invaluable support on explaining the complexities of the Triana code.

This work was possible due a grant from the Agentcities.NET project, to which we would also like to thank.
References


