Pattern/Operator Based Problem Solving Environments

Cecilia Gomes[1], Omer F. Rana[2], and Jose C. Cunha[1]

[1] CITI Center, University Nova de Lisboa, Portugal
[2] School of Computer Science, Cardiff University, UK

Abstract. Problem Solving Environments (PSEs) provide a collection of tools for composition of scientific applications. Such environments are often based on graphical interfaces that enable components to be combined, and in some cases, subsequently scheduled on computational resources. A novel approach for extending such environments with Design Patterns and Operators is described – as a way to better manipulate the available components – and subsequently manage their execution. Users make use of these additional abstractions by first deploying 'Structural Patterns' and by refining these through 'Structural Operators'. 'Behavioural Patterns' may then be used to define the control and data flows between components – subsequent use of 'Behavioural Operators' manage the final configuration for execution control and dynamic reconfiguration purposes. We demonstrate the implementation of these Patterns and Operators using Triana [14] and the Distributed Resource Management Application (DRMAA) API [10].

1 Introduction and Motivation

A Problem Solving Environment (PSE) is a complete, integrated computing environment for composing, compiling, and running applications in a specific area [1]. In many ways a PSE is seen as a mechanism to integrate different software construction and management tools, and application specific libraries, within a particular problem domain. One can therefore have a PSE for financial markets [4], for Gas Turbine engines [5], etc. Focus on implementing PSEs is based on the observation that previously scientists using computational methods wrote and managed all of their own computer programs – however now computational scientists must use libraries and packages from a variety of sources, and those packages might be written in many different programming languages. Engineers and scientists now have a wide choice of computational modules and systems available, enough so that navigating this large design space has become its own challenge. A survey of 28 different PSEs by Fox, Gannon and Thomas (as part of the Grid Computing Environments WG) can be found in [6], and practical considerations in implementing PSEs can be found in Li et al. [2]. Both of these indicate that such environments provide "some backend computational resources, and convenient access to their capabilities". Furthermore,

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 964–971, 2004.

workflow features significantly in both of these descriptions. In many cases, access to data resources is also provided in a similar way to computational ones. Often PSE and Grid Computing Environment is used interchangeably – as PSE research predates the existence of Grid infrastructure. The aim of our work is to: (1) extend the capabilities of existing PSEs with support for Patterns and Operators, and (2) to enable the management of such applications subsequently by mapping "Behavioural" patterns and operators to a resource management system. Patterns allow the abstraction of common interactions between components, thereby enabling reuse of interactions that have proven useful in similar domains. 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. Users may also define new patterns found to be useful, and add these as components for use by others. Patterns and Operators also provide additional capability that is not easily representable via visual components. Our approach treats patterns as first class entities but differs from other works [7, 8] in that the user may explicitly define structural constraints between components, separately from the behavioural constraints. Our approach is somewhat similar to that of van der Aalst et al. [17] – although they do not make a distinction between structural and behavioural patterns. They also focus on Petri net models of their patterns, whereas our concern is to link patterns with particular resource managers and composition tools. The approach presented here is primarily aimed at computational scientists and developers, who have some understanding of the computational needs of their application domain. A scientist should be aware about the likely co-ordination and interaction types between components of the application (such as a database or numeric solver etc). The structural and behavioural patterns presented here will enable such scientists and developers to utilise common usage scenarios within a domain (either the use of particular components, such as database systems, or interactions between components, such as the use of streaming). Section 2 introduces our concept of Patterns and Operators, and Section 3 demonstrates how these are implemented in the Triana (the workflow system for the European GridLab project [9]) – and describe theme (1) mentioned above. Theme (2) is then explained in Section 4.

2 Structured Composition of Applications in PSEs Based on Grids

Structural Pattern Templates encode component connectivity, representing topologies like a ring, a star or a pipeline, or design patterns like Facade, Proxy or Adapter [15]. The possibility of representing these structural constraints allows, for example, the representation of common software architectures in high-performance computing applications. For example, the pipeline pattern may be used in a signal processing application where 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. The proxy pattern, for instance, allows the local presence of an entity's

surrogate, allowing access to the remote entity. Grid services, for example, are usually accessed through a proxy (or gatekeeper).

Behavioural Pattern Templates capture recurring themes in component interactions, and define the temporal and the (control and data) flow dependencies between the components. Generally, these applications involve distribution of code from a master, the replication of a code segment (such as within a loop), or parameter sweeps over one or more indices. We provide several behavioural patterns such as Master-Slave, Client-Server, Streaming, Peer-to-Peer, Mobile Agents/Itinerary, Remote Evaluation, Code-on-Demand, Contract, Observer/Publish-Subscriber, Parameter sweep, Service Adapter, and so on. For example, the Service Adapter pattern "attaches additional properties or behaviours to an existing application to enable it to be invoked as a service" [16]. The Master-Slave pattern, in turn, can be mapped to many parallel programming libraries, and represents the division of a task into multiple (usually independent) sub-units – and shares some similarities with the Client-Server pattern – although the control flow in the latter is more complex.

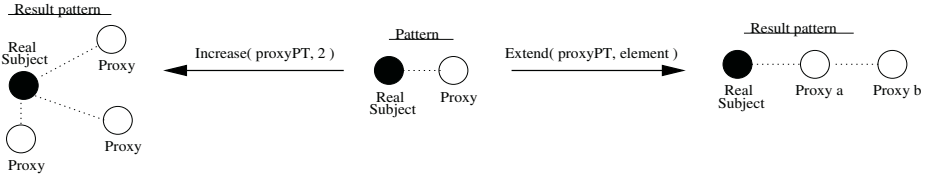

Fig. 1. The increase and extend structural operators.

Structural Operators support the composition of structural patterns, without modifying the structural constraints imposed on the pattern. This provides a user with a simple and flexible way to refine structural patterns. There are several structural operators such as increase, decrease, extend, reduce, rename, replace, replicate, embed, etc. For example in figure 1 it is possible to observe the result of applying the increase and extend operators to the Proxy pattern.

Behavioural Operators are applied over the structural operator templates combined with the behavioural patterns after instantiating the templates with specific runnable components. Behavioural operators act upon pattern instances for execution control and reconfiguration purposes. Behavioural operators include: Start (starts the execution of a specific pattern instance), Stop (stops the execution of a pattern instance saving its current state), Resume (resumes the execution of a pattern instance from the point where it was stopped), Terminate (terminates the execution of a specific pattern instance), Restart (allows the periodic execution of a pattern instance), Limit (limits the execution of a specific pattern instance to a certain amount of time; when the time expires the execution is terminated), Repeat (allows the repetition of the execution of a specific pattern a certain number of times), etc. Both structural operators and behavioural operators can be combined into scripts which may be later reused in similar applications.

3 Implementation over the Triana GCE

A prototype has been implemented by extending Triana [14], and allows developers to utilise a collection of pre-defined patterns from a library. Triana comes with components (called *units*) for signal processing, mathematical calculations, audio and image processing, etc, and provides a wizard for the creation of new components, which can then be added to the toolbox. Structural Patterns appear as standard components that can be combined with other patterns or executable units. Triana provides both a composition editor, and a deployment mechanism to support this. The Pattern library provided within Triana treats patterns as "group units" (i.e. units made up of others). Each element within such group units is a "dummy" component (or a place holder) and can subsequently be instantiated with executables from the Triana toolbox. Hence, structural pattern templates are collections of dummy components that can be instantiated with other structural pattern templates or with executables.

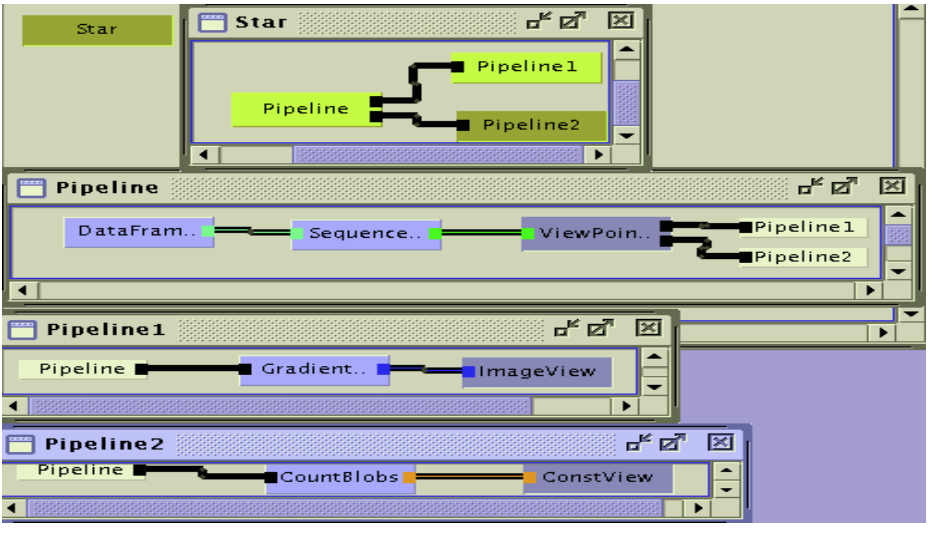

Fig. 2. A possible final configuration for the image processing of the "Galaxy Formation example".

A Galaxy simulation application with Triana is illustrated in figure 2. The Galaxy formation example may be represented by a star pattern template, where the nucleus contains the actions necessary to generate and control the animation execution, and the satellites represent image processing and analysis actions. Both the actions at the nucleus and at the satellites are supported by pipeline templates. As such, the *Pipeline* pattern instance shown in the figure, represents the actions at the nucleus, and consists of three stages. *Pipeline* is connected to the two satellites, namely, *Pipeline1* and *Pipeline2*, and produce data to these

two pattern instances. *Pipeline1* is embedded in one of the satellites and connects two units for image processing. *Pipeline2* binds two units that together support analysis of the data produced at the nucleus. See [12] for details.

4 Mapping to the DRMAA API

Behavioural patterns 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. We map behavioural patterns over DRMAA [10]. Pattern execution essentially involves coordination between modules. Execution is therefore delegated via DRMAA to third party resource management systems (DRMAA provides a generalised API to execute jobs over Distributed Resource Management Systems (DRMSs)). DRMAA includes common operations on jobs like termination or suspension. A job is a running application on a DRMS and it is identified by a *job_id* attribute that is passed back by the DRMS upon job submission. This attribute is used by the functions that support job control and monitoring. DRMAA API uses an IDL-like definition (with IN defining an input parameter, OUT defining an output parameter, and INOUT defining a parameter that may be changed), and also provides support for handling errors (via error codes).

To configure and execute an application using the patterns library, a user (developer) needs to undertake the following:

- A structural pattern – such as a "pipeline" – is selected from the patterns library. This appears as a standard Triana (group) unit. Figure 2 illustrates a number of different pipeline instances.
- A user may add or remove elements from the structural pattern chosen. This is achieved by using structural operators such as "increase" or "decrease" respectively.
- A user may now select a behavioural pattern – such as "dataflow" – to indicate how interaction between the units/elements is to take place.
- An entity at the pattern level is defined, the *pattern executor*, responsible for enforcing the selected behavioural pattern at each element.
- All component place-holders are instantiated with components (*Applications*) that may represent a unit in Triana or a group of units organized in a workflow.
- A user may now wish to use a behavioural operator – such as "start" or "stop" – on the behavioural pattern. These operators are supported by functions in the DRMAA API that manage the execution of the *Applications* by a resource manager. The execution of each *Application* is supported by a job (running executable) in the resource manager.

A user therefore may select structural patterns/operators followed by behavioural patterns/operators – all of which are implemented in Java. It is important to note that behavioural patterns/operators can only be applied to structural patterns – and not to arbitrary Triana units. A user does not need to know the

actual implementation of any of these patterns/operators to make use of them – as they are primarily pre-defined group units in Triana or scripts. We therefore do not expect the user to be familiar with any particular programming language or scripting tool. Experienced developers, however, may add their own operators or patterns to our library.

Application execution using DRMAA requires the definition of attributes like the application's name, its initial input parameters, the necessary remote environment that has to be set up for the application to run, and so forth. These attributes are used to explicitly configure the task to be run via a particular resource manager. Although DRMAA has the notion of sessions, only one session can be active at a time. A single DRMAA session for all the operators is assumed. Hence `drmaa_init` and `drmaa_exit` routines are called, respectively, after the pattern instance is created and in the end of the script program. As an example, we show how a pipeline pattern can be mapped to DRMAA: Element `pattern_elements[MAX_ELEMS]` – contains the *Elements* that compose a specific pattern instance. Similarly, `job_identifiers[MAX_ELEMS]` represents the identifiers returned by the `drmaa_run_job` routine for jobs created to support `pattern_elements`. The order of the activities is preserved. DRMAA variables frequently used: `INOUT jt` is a job template (opaque handle), and `INOUT drmaa_context_error_buf` contains a context-sensitive error upon failed return. The examples are illustrated in a Java-like notation.

Start Operator – to initiate execution of Pipeline Elements.

```
/* launch all activities in the pipeline */
for( int index = Pipeline.pattern_elements.length -1 ; index >= 0;
index -- ) {
   int ret = drmaa_allocate_job_template( jt, drmaa_context_error_buf );
   process_error( ret, drmaa_context_error_buf );
   define_attributes( jt, Pipeline.pattern_elements[index] );
   /* Pipeline.startTime defines the time at which all
   elements in the pipeline instance should start running.  */
   ret = drmaa_set_attribute( jt, drmaa_start_time,
                              Pipeline.startTime,
                              drmaa_context_error_buf );
   process_error( ret, drmaa_context_error_buf );
   /* run one job at the specified time */
   ret = drmaa_run_job( job_id, jt, drmaa_context_error_buf );
   process_error( ret, drmaa_context_error_buf ); }
```

Repeat Operator – in this instance a single operator is used to re-execute an entire pattern instance a certain number of times ("n" in the code).

```
for( int count = 0; count < n; count++ ) {
   Start( Pipeline );
   /* wait for all the jobs that compose the pipeline to terminate */
   drmaa_synchronize( Pipeline.job_identifiers, timeout, 0,
                  drmaa_context_error_buf );
   /* timeout is bigger than all jobs' execution times */ }
```

5 Conclusions and Future Work

The extension of a Problem Solving Environment (Triana) with Patterns and Operators is described. Composition is achieved using a pattern extended graphical interface provided with Triana – whereas execution is managed by mapping Operators to the DRMAA API. We believe a Pattern based approach is particularly useful for reuse of component libraries in PSEs, and for mapping applications constructed in PSEs to a range of different execution environments. The DRMAA API was selected because of the significant focus it initially received within the Grid community – and the availability of commercial resource management systems (such as Grid Engine from Sun Microsystems) that make use of it. We are also investigating alternatives to DRMAA (such as Java CoG) [11] – primarily as current versions of DRMAA are aimed at executing batch jobs. With the emerging focus on Web Services in the Grid community, the DRMAA API has also lagged behind other equivalent developments (such as the Java CoG kit).

Patterns provide a useful extension to existing PSEs, as they enable the capture of common software usage styles across different application communities. The pipeline and star structural patterns, for instance, are commonly found in scientific applications (such as integrating a data source with a mesh generator, followed by a visualiser). Describing such compositions in a more formal way (as we have attempted to do here), will enable practitioners in the community identify common software libraries and tools. This is particularly important as software that performs similar functionality is available from a variety of different vendors. Providing the right balance between tools that require users to possess programming skills, and those that are based on a visual interface is difficult to achieve. By combining the visual interface of Triana with more advanced patterns and operators, we are attempting to enhance the functionality offered through (a variety of) existing workflow tools. Full usage of these ideas by the applications community is still a future aim for us.

References

1. E. Gallopoulos, E. Houstis and J. Rice, “Computer as Thinker/Doer:Problem-Solving Environments for Computational Science”, IEEE Computational Science and Engineering, 1(2), 1994.
2. M. Li and M. A. Baker, “A Review of Grid Portal Technology”, to appear in Book, “Grid Computing: Software Environment and Tools” (ed: Jose Cunha and O.F.Rana), Springer Verlag, 2004
3. J. Novotny, M. Russell and O. Wehrens “GridSphere: A Portal Framework for Building Collaborations”, 1st International Workshop on Middleware for Grid Computing (at ACM/IFIP/USENIX Middleware 2003), Rio de Janeiro, Brazil, June 2003. See Web site at: `http://www.gridsphere.org/`. Last visited: January 2004.
4. O. Bunin, Y. Guo, and J. Darlington, “Design of Problem-Solving Environment for Contingent Claim Valuation”, Proceedings of EuroPar, LNCS 2150, Springer Verlag, 2001.

5. S. Fleeter, E. Houstis, J. Rice, C. Zhou, and A. Catlin, "GasTurbnLab: A Problem Solving Environment for Simulating Gas Turbines", Proceedings of 16^{th} IMACS World Congress, 104-5, 2000.
6. G. Fox, D. Gannon and M. Thomas, "A Summary of Grid Computing Environments", Concurrency and Computation: Practice and Experience (Special Issue), 2003. Available at:http://communitygrids.iu.edu/cglpubs.htm
7. B. Wydaeghe, W. Vanderperren, "Visual Composition Using Composition Patterns", Proc. Tools 2001, Santa Barbara, USA, July 2001.
8. ObjectVenture, The ObjectAssembler Visual Development Environment. See Web site at: `http://www.objectventure.com/objectassembler.html`. Last visited: March 2003.
9. The GridLab project. See Web site at: `http://www.gridlab.org/`. Last visited: January 2004.
10. Habri Rajic, Roger Brobst et al., "Distributed Resource Management Application API Specification 1.0". Global Grid Forum DRMAA Working Group. See Web site at: `http://www.drmaa.org/`. Last visited: September 2003.
11. Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane, "A Java Commodity Grid Kit," Concurrency and Computation: Practice and Experience, vol. 13, no. 8-9, pp. 643-662, 2001, http:/www.cogkits.org/.
12. M.C.Gomes, O.F.Rana, J.C.Cunha "Pattern Operators for Grid Environments", Scientific Programming Journal, Volume 11, Number 3, 2003, IOS Press, Editors: R. Perrot and B. Szymanski.
13. M.C.Gomes, J.C.Cunha, O.F.Rana, "A Pattern-based Software Engineering Tool for Grid Environments", Concurrent Information Processing and Computing proceedings, NATO Advanced Research Workshop, Sinaia, Romenia, June 2003, IOS Press.
14. I. Taylor et al., "Triana" (`http://www.trianacode.org/`). Triana is the workflow engine for the EU GridLab project (`http://www.gridlab.org/`). Last Visited: January 2004.
15. E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.
16. O. F. Rana, D. W. Walker, "Service Design Patterns for Computational Grids", in "Patterns and Skeletons for Parallel and Distributed Computing", F. Rabhi and S. Gorlatch(Eds), Springer, 2002.
17. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Distributed and Parallel Databases, 14(3), pages 5-51, July 2003