Abstract—Software patterns are reusable solutions for repeated software design problems. They are applied on object oriented software systems design. The patterns can be represented by suitable Unified Modeling Language - UML diagrams showing the structure of their class system and the interactions and relations amongst the class system members. Patterns can be identified by the solutions they offer in certain problems of certain contexts and can be grouped in categories according to the solutions they offer to broader problem and context areas. The two main categories are the architectural and the design patterns. The later consists of subcategories including fundamental, creation, structural, behavioral, concurrency and partitioning patterns. The individual patterns can be combined so as to offer complex solutions when necessary. Software patterns, which can be applied in the design of object oriented systems running on multiprocessors or cluster environments, are investigated in this paper.

Index terms – high performance computing, concurrency, software design, software patterns,

I. INTRODUCTION

Architectural and design patterns are standardized solutions to commonly and frequently occurring software design problems [11]. When the solutions are used for design problems solving during the design phases of the software life cycles, they can both deter the appearance of important problems and ensure that the under development object oriented code will be readable, comprehensive and easily maintainable.

Most of the patterns have already been investigated [23]. They can be separated in categories, which are frequently used nowadays. Object oriented software patterns have been utilized for the design of applications that support many areas and fields of sciences, such as Biology, Biomedicine, Web applications, Communications, etc.

Design patterns have been used to improve [13] the quality of communication software systems. Patterns have also been proposed to solve design problems in the area of high performance computing - HPC [14], concurrent programs and distributed applications [24], [25], [3], [2].

Applying patterns to concurrent systems design can reduce software development time, improve code maintainability, and increase code reuse over traditional software engineering techniques [15]. Patterns can also be applied in the design of object oriented software systems running on multiprocessors or computer clusters.

High performance computing has been an area of interest for reverse engineering of design patterns [5]. In spite of the increased popularity of pattern based design of parallel, distributed and concurrent systems, the existing patterns cannot cover entirely the needs for solving complicated high performance software design problems. In other words several of such systems, whose design is based on patterns, face serious problems such as limited flexibility and zero extensibility [16].

High performance computing faces complicated problems, which the existing design pattern based frameworks and techniques cannot address [6], [16], [2]. In this paper, combinations of patterns (hybrid patterns), which directly aim to assist designers to build better high performance systems, are investigated.

These hybrid patterns, combine concurrency patterns with other patterns from different categories in order to solve complicated concurrent problems and give the opportunity to high performance systems to become more reliable. In particular, two hybrid patterns, which combine concurrency patterns with design patterns, are defined. The first pattern combines Scheduler and Proxy patterns, aiming to reduce the resources needed for the system. The second attempts to ensure that many users can access different data without errors. It combines Read - Write Locking and Strategy patterns.

This paper is divided in 4 sections, which are organized as follows: Section 2 provides the background and the related literature review for the design patterns used in this paper. Section 3 describes the concurrency patterns and the related literature. In section 4 we describe the proposed hybrid patterns. Finally, in section 5 the findings are summarized as conclusions.

1Dept. of Informatics with Applications in Biomedicine, University of Central Greece, 2-4 Papasiopouloy Str, Lamia 35100, Hellas
sfount@phs.uoa.gr

*Correspondence Address: POBox 4176, Athens 10210, Hellas.
II. DESIGN PATTERNS

A. The Strategy Pattern

The strategy pattern belongs to the Behavioral patterns category and is very useful in applications, which need to change dynamically their behavior [19]. Behavioral patterns determine how the objects in a pattern communicate and provide the designers flexibility for this communication [22]. It decouples a procedure from its host and encapsulates it in a separate class. In other words, an object and its behavior are separated from each other and exist in different classes. This offers the option for procedure alteration at any time. One benefit from this is that, if one object needs to have several behaviors, it is much easier to be observed if their behaviors are implemented in different classes than in the same class. More specifically, if the object needs to add, remove or alter any of its behaviors, this can easily be accomplished when the behaviors are designed and implemented as different methods of different classes. The behaviors are called strategies and are implemented in different subclasses that inherit the same abstract class. The class diagram of the strategy pattern is shown in the following Figure 1.

![Figure 1: The class diagram of the strategy pattern is illustrated.](image1)

Garbinato et. al [17] have showed how the Strategy pattern has been used to build BAST1, an extensible object oriented framework for programming reliable distributed systems. The Strategy pattern, in the above-mentioned work, allows the limitations of inheritance to be overcome, when trying to compose protocols. Basing the BAST framework on the Strategy pattern had the advantage of making it easily extensible. The Strategy design pattern has also been proposed in the area of self-adaptive systems [18]. This design pattern provided support for the analysis of existing self-adaptive systems. It was also used during the design of a self-adaptive system in order to identify the issues and the needed tools.

B. The Proxy Pattern

The Proxy pattern belongs to the Fundamental patterns and is used when something complicated needs to be represented as something simpler. Despite of the fact that the Fundamental patterns are so frequently used, that sometimes are not ever referenced, the Proxy pattern is one of the most under-utilized yet extremely useful patterns available. When not all the methods from the objects, having heavy method-use costs, need to be used, then it is not worth it to use them. They will be used in case they are necessary as a whole. The solution is to use lightweight objects, which would have the same interface with the heavy objects. The lightweight objects are named proxies and instantiate the objects, having heavy method-use costs, whenever necessary [20].

![Figure 2: The class diagram of the Proxy pattern is shown.](image2)

An extension of the Proxy Pattern, which abstracts the true server from the client, has been proposed [21]. This separates the client and server, allowing the hiding of specified properties of the server from the clients. In [26] a design pattern is presented called Distributed Proxy, for distributed object communication.

III. CONCURRENCY PATTERNS

These patterns present general solutions in frequent concurrency problems that can be found in parallel or distributed systems. They focus on two types of problems: in the distribution of resources, focusing in the deadlock management and the simultaneous implementation of processes. These processes should follow a correct specific order. The choice of concurrency architecture has an important impact on both planning and performance of complicated multithreading applications. No concrete concurrency architecture is suitable for all cases and all platforms of hardware and software. In software technology these patterns are types of design patterns, which examine multithreading planning. The most common of these patterns are Active Object, Balking Pattern, Double checked locking Pattern, Guarded suspension, Leaders/Followers Pattern, Monitor Object, Read write lock Pattern, Scheduler Pattern, Thread pool Pattern, Thread-Specific Storage Pattern and Reactor Pattern.

A. The Scheduler Pattern
In high performance computing, it is important to protect shared resources, while multiple requests and update processes attempt to access the data concurrently, risking performance impacts, dead-locks, data corruptions etc. Concurrency design patterns aim on alleviating the problems mentioned above.

The scheduler pattern is a concurrency pattern. It is used to schedule the times that the threads will execute pieces of their code. It uses an object that puts in line threads that wait to run. It provides a mechanism in order a policy of planning to be applied. The policy is encapsulated in its class and it is reusable. In [9] the scheduler pattern is used for designing space exploration of real time systems.

Figure 3: The class diagram of the Scheduler pattern is illustrated.

Two positive consequences of the Scheduler pattern are the following. First, this pattern provides a way to check, when the threads can execute a piece of code. Second, the schedule policy is encapsulated in its class and is reusable.

B. The Read-Write Lock Pattern

The RWL (Read-Write Lock) pattern is a software design pattern, which allows the simultaneous access of reading in an object but requires exclusive access for the methods of writing. In this pattern, a lot of readers can read the data in parallel, but an exclusive locking is required when data will be written. When a writer writes the data object, the readers will be prevented until the writing is finished.

This pattern is used in [8] in order to implement efficient and predictable dispatching components for multithreading systems. It has several different forms. One of these is the strategized locking pattern [4], which increases the flexibility and the maintainability.

Any number of threads can simultaneously get the information of the Data object, provided that no thread sets information, at the same moment on the same object.

Figure 4: Class diagram of the Read-Write Lock pattern is illustrated.

Its set methods should be used one at a time, while no other different data operation is executed. The Data objects should coordinate the set and get methods, so that they obey to the restrictions.

The Read-Write Lock pattern increases a special type of synchronization of its methods and achieves a corresponding special mutual exclusion. It also allows the re-use of logic of synchronization control, increasing the concurrent access whenever more reading than writing activities exist. Nevertheless, this pattern is not applicable whenever the number of processes of writing is greater than the number of processes of reading [1].

IV. PATTERN COMBINATIONS

A. The Scheduler - Proxy Pattern

Figure 5: The class diagram of the combination of the Scheduler and the Proxy patterns.

This combination is a solution for the problem of independency of Client and the classes it uses, from these classes objects process access. At the same time the execution of the services offered are scheduled.

The client does not reference to the real object it wishes to use. It references to an Interface, and has access to the proxy object that implements this interface. In other words the client despite of the fact that has no reference to the real object, it has a reference to proxy interface through which
implement the Concrete Strategies provide different behaviors and also get methods, so that they obey to the restrictions. The subclasses. Strategy objects should coordinate their set and same time. Set methods should appear one per implemented Strategy object, only if not thread threads can simultaneously that inherit and implement this interface. Any number of threads can be more than required.

Instances of Scheduler classes schedule the Request objects. A Scheduler class does not know anything about the Request class that it creates. It has access to the Request objects through the interface Queue Scheduling, which they implement. A Scheduler class decides when the next request will run. However, it is not responsible for the order in which the requests will be executed.

The Request objects implement the Queue Scheduling interface. This interface serves two tasks: first, by referring to a Queue Scheduling interface, the Processor classes avoid dependency to a Request class, second, by calling the methods, which are defined by the Queue Scheduling, interface the Scheduling classes are able to decide which Request object will be processed next. That increases the reusability of the Scheduling classes.

As a result, the benefits from the Scheduler and the Proxy pattern can exist together and support the high performance computing in specific areas. If the user needs the service to be performed, then the Scheduler pattern will carry out its task. If a part of the service is needed and not the whole service, then the Proxy pattern is applied and time cost can be reduced. Therefore the way for optimisation of the workings of such systems is opened.

Without this combination of patterns, the Scheduler pattern would have unnecessary redundant processing. Another benefit, which is very important to high performance systems, is that the resources that would be used in case the Scheduler acted alone without the Proxy cooperation would be more than required.

### B. The Read Write Lock – Strategy Pattern

As it can be seen in figure 6, the Strategy interface has set and get methods for the data encapsulated in the instances that inherit and implement this interface. Any number of threads can simultaneously get the information from a Strategy object, only if no thread sets information at the same time. Set methods should appear one per implemented subclasses. Strategy objects should coordinate their set and get methods, so that they obey to the restrictions. The Concrete Strategies provide different behaviors and also implement the set and get methods inherited from the interface.

The client interacts with the Strategy interface, but it does not know the real class of the object or how the class implements the methods. The interface is common to every strategy. Some consequences are that this pattern allows one of the strategies to be chosen dynamically and thus there is flexibility. Moreover, the exclusive write locking ensures that there will not be any competition if one user writes and some other user reads or writes information. Many readers can use fearless the information adding that there is less conflict between the resources [7], [10].

The get methods of an object don’t return any information until they take a read lock. Every read-write Lock object is connected with a Strategy object. While the thread has a read lock, the get method is sure that is safe to take data from the object. If the write locks haven’t finished their task, when the readLock method of the read-write Lock object is called, nothing is returned until all write lock methods have completed their task. On the other hand, the calls to the readLock method return data [7]. When a get method has finished its task receiving data from the object, calls the done method from the read-write Lock object. This call forces the running thread to stop the read lock [1].

This second combination gives some very crucial advantages to the design of high performance computing. The first benefit is that many clients can use the Strategy interface and have access to data. This access includes setting and receiving information. This data contains different information because of the different strategies. The second advantage is that with this combination the errors that would occur have been minimised through the locking. The clients can simultaneously access without having to consider the data corruption that would appear when two clients write at the same time.

### V. CONCLUDING REMARKS

In this paper two hybrid patterns are discussed. The first is a combination of Proxy and Scheduler patterns, while the second is a combination of Strategy and Read-Write Lock
patterns. Considering the increased complication of high performance computing systems, these patterns can be used for the development of frameworks or tools supporting the design and implementation of high performance computing systems.

The herein studied hybrid patterns can be further combined with other patterns to solve more complicated design problems related to HPC.

REFERENCES


[13] Douglas C. Schmidt Paul Stephenson , Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms, Department of Computer Science Ericsson, Inc. Washington University, St. Louis, MO 63130 Cypress, CA 90630


