World Academy of Science, Engineering and Technology
International Journal of Industrial and Manufacturing Engineering
Vol:4, No:2, 2010

Aspect based Reusable Synchronization Schemes

Nathar Shah

***Abstract*—**Concurrency and synchronization are becoming big issues as every new PC comes with multi-core processors. A major reason for Object-Oriented Programming originally was to enable easier reuse: encode your algorithm into a class and thoroughly debug it, then you can reuse the class again and again. However, when we get to concurrency and synchronization, this is often not possible. Thread-safety issues means that synchronization constructs need to be entangled into every class involved. We contributed a detailed literature review of issues and challenges in concurrent programming and present a methodology that uses the Aspect-Oriented paradigm to address this problem. Aspects will allow us to extract the synchronization concerns as schemes to be "weaved in" later into the main code. This allows the aspects to be separately tested and verified. Hence, the functional components can be weaved with reusable synchronization schemes that are robust and scalable.

***Keywords*—**Aspect-orientation, development methodology software concurrency, synchronization.

I. Introduction

DESIGNING classes in object-oriented paradigm for concurrent applications require considerable effort to ensure they are thread-safe and yet safe from liveness hazards. In single threaded programs, performance improvements are made in various ways: the execution instructions can be reordered, and frequently used data cached in registers or processor-local caches [1]. These improvements, if applied to multi-threaded programs, however, will undermine safety and consistency of the applications. Different threads seeing different cached value for the updates done by other threads, for example, will result in a race condition. It is a condition where the correctness of an operation depends on the relative timing or interleaving of multiple threads at the run-time.

The objective of this paper is to motivate readers on the need to develop a new methodology for concurrent programming. We also present a new methodology based on aspect-orientation to separate synchronization aspects from an object-based program into synchronization schemes, so that tested synchronization schemes can be reused by other objects that have the same patterns of synchronization, therefore contributing to robust programs.

The paper is organized as follows: in section II, we focus on the issues and challenges in building Java concurrent applications and along the way set the foundations to separate synchronization aspects of the concurrency as a concern that can be reused; Section III presents the methodology that we propose; and section IV looks into some relevant related works.

Nathar Shah is with the Faculty of Information Technology, Multimedia University, Cyberjaya, 63100 Selangor, Malaysia (phone: +603-8312-5230; e-mail: nathar.packier@mmu.edu.my).

II. Issues and Challenges in Building Concurrent Applications

A. Deadlock

Deadlock is a term given to a scenario of cyclic dependency between resources hold by the holder and the requester. In multi-threaded Java applications, deadlocks can be of two types: lock-ordering deadlock, and resource deadlock [1]. The commonality between them is that they occur when locks or resources are shared. When the shared locks or resources form cyclic dependency or infinitely blocked, deadlock occurs. Contributing factors in lock-ordering deadlock include operations with *coarse grained synchronization that may invoke alien methods* which consequently acquire separate locks, and also *unregulated lock sharing*. Noteworthy abstraction concepts like classes and interfaces enables dynamic binding of reference variables to an object at run-time, therefore also enables dynamic lock order deadlock. Brian Goetz et al. highlighted two mechanisms to prevent deadlock: first, applications that need to acquire multiple locks need to observe a *global locking order* and second, *using open calls* - invoking methods with no lock held.

Currently no mechanism is available to enforce global locking order policy. Similarly restricting open calls requires careful design. However, both greatly improve the liveness analysis of a program. Clearly, at the outset, invoking alien methods that may acquire separate locks, unregulated lock sharing, global locking order, and open calls are related to the issue of locking and may contribute or prevent deadlocks. Hence, the key is in the ability to effect coordination in possessing and releasing of locks.

Having a locking protocol to coordinate between the synchronization schemes and functional components will relieve programmers from human prone errors and result in a more reliable program. An optimistic solution to the problem is to have synchronization patterns that can be weaved to functional components programmed in a pseudo-standard way. On the other hand, compare-and-swap (CAS) atomic variables - used in non-blocking algorithms and lock-free algorithms - however, are an alternative to current thread unsafe and liveness hazard prone lock-based algorithms. But, experiments have shown that at low contention, CAS performs better and at high contention, locking mechanism performs better [1]. Furthermore, although CAS based atomic variables have both atomicity and happens-before relationship, they are unable to account for invariants in the program. Therefore, thread-safety is still an issue with CAS based atomic variables.

B. Starvation

Avoiding starvation is another challenge in developing

multi-threaded applications. Starvation occurs when a thread is perpetually denied access to resources it needs in order to make progress. In the worst case, when cyclic dependency between starving threads is formed, it leads to thread-starvation deadlock [1]. Thread priorities and granularity of synchronization/locking are the main lead factors of starvation. Greedy higher priority threads will always pre-empt and run on top of the lower ones leaving others starving to get CPU cycles to complete their processes. Whereas in coarse grained synchronization/locking, it prevents other threads waiting for the lock from progressing for the period of locking – lock starvation.

CAS based atomic variable implementation fare better in this respect, however, still susceptible to starvation if the thread priority decides. The CAS atomic variables, on the other hand, are not compatible with coarse grained granularity in the presence of invariants. They result in thread-safety issues. But, coarse grained locking is good to protect the invariants. For independent state variables, a multiple lock guard will be able to improve the scalability – as a result of less thread contention. A combination of both CAS atomic variable and locking approach will give the most benefit.

C. Livelock

Livelock also disrupts the progress of threads. However, the threads will not be blocked as in deadlock and starvation, but will strive to complete an operation that will always fail. This case is prevalent in CAS based non-blocking-algorithms and lock-free- algorithms in a highly contentious environment. A CAS update to a variable is hoped to be successful and if not the detection mechanism will allow the caller to handle by retrying, backing off, or giving up. The retying may always fail if some other thread has done the required changes.

D. Thread Safety

Thread safety is required when multiple threads access a shared mutable state. Brian Goetz[1] specified three base elements in the design of a thread-safe class: the variables that forms the object's state, invariants that constraint the state variables, and the policy to manage concurrent access to the object's state. Primary means to achieve thread safety are by using synchronization, locking, and atomic variables. In addition, immutable objects, and thread confined objects may also be used. A program composed of thread-safe classes may not necessarily be classified as thread-safe. Vice-versa, a thread-safe program may contain non thread-safe classes also. What matters is if the invariants in the program are protected. In the presence of invariants, however, synchronization and locking approaches are more suitable as they protect the coherence of the invariants. With either blocking or non-blocking synchronization, race conditions can be prevented. A blocking synchronization will obtain an intrinsic lock and therefore serialize the execution. However, non-blocking synchronization relies on optimistic approach such as CAS in guaranteeing atomicity and happens-before relationship.

Apart from the synchronization issue, there are also other concerns for thread safety such as safe publication. Publishing an object, an act of making it available to its enclosing environment (i.e. outside its scope) via means like returning a reference from a non-private method, passing objects as parameters to a method, publishing inner class instance, or storing reference in a public static field has the possibility of making visible semi constructed object states. The object is said to be escaped. An escaped object can compromise the thread safety in that it allows other threads to change its invariant and also states illegally. These are the minute issues a programmer hardly realize or remember. In essence, objects need to go through some form of pre-conditions and post-conditions when its methods are called. A right mechanism is one that allows misconstruction to be checked during the pre-test process and corrective action performed before executed.

III. A Methodology for Reusable Synchronization Schemes

Earlier, major challenges in building concurrent programs are stated. Among others, it defined the challenges, elaborated on the causes, examined approaches to solutions, and outlined issues to be considered for a reusable mechanism. The trend is a coordination mechanism between the functional components and the synchronization schemes/patterns via some protocol where the concurrency schemes' invariants are guarded via pre and post conditions. The protocol should define the configuration of the functional components such that the synchronization scheme can be weaved to the identified configuration points. The pre and the post conditions in the general sense is a monitor for communication with the objects such that it will be able to enforce invariants, optimize implementation of invariants (e.g. by using CAS for non-interfering conditional variables), and prevent escaping of objects. Current state-of-the-art in the field (Java Language), however, emphasize on providing concurrent APIs like for concurrent data structures – with taken care thread safety - and is expected to be used correctly with the functional components.

Aspects [2] allow us to extract the synchronization concerns to be "weaved in" later into the main code. Hence, aspects can be separately tested and verified. The idea is depicted in Fig. 1 overleaf. In Fig. 1 (A), functional component and synchronization schemes are tangled. In the new model, as represented in Fig. 1(B), functional component and synchronization schemes are separated and are weaved in as per coordination protocol defined in the aspect by the aspect weaver.

The methodology to achieve the aspect based concurrent programming model is as follows:-

1) Classify and generalize synchronization concurrency controls

Patterns on how synchronization concurrency controls affect functional components need to be identified and generalized. These patterns will set requirements for the kinds of joint points and restructuring expected at the functional components. The join points together with the protocol to be defined will enable synchronization aspects to be weaved separately by using the AspectJ[2] weaver. Then,

classification of these patterns will allow implementation of simple synchronization concurrency control units which will enable reuse. These synchronization units will then be composed to form synchronization schemes.

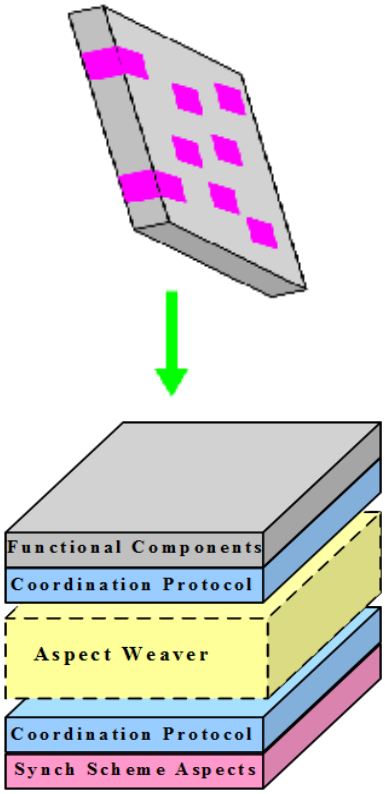

Fig. 1 (a) Represents current concurrent programming model and (b) Aspects based programming model

2) Specify and design the coordination protocol

In order for both the functional components and the synchronizing aspects to work in tandem and seamlessly, a protocol for their coordination are specified. In the earlier work of [3] and [4], protocols were developed for thread access to functional components concurrently. The protocol is developed by structuring the functional components so that appropriate join points for aspects can be made available for synchronization aspects to be weaved with the semantics provided by the protocol.

3)Develop aspects for the synchronization schemes

The implemented synchronization concurrency units are composed to form synchronization schemes which can be reused. The scheme are developed using AspectJ[2]. The pointcuts for the aspect are determined for the join points revealed earlier.

IV. Related Works

Two of the interesting views are: reforming how concurrent programs are developed by introducing new sets of language constructs and guarding objects against concurrency-related problems. The D Language Framework [4], for example, focuses on separating the synchronization aspects, the distribution aspects, and the functional components. In that framework, two languages – COOL and RIDL – were designed to tackle the first two aspects. They were glued together with the functional component of a class by an aspect interface contained by the class.

Our interest was in the COOL language. It offers syntactic constructs to specify synchronization states, mutual exclusion of methods, self-exclusion of methods, and a method manager for guarded suspension and notification of threads. These language constructs are all contained in a construct called a coordinator which can reference to an instance of a class, or any class.

Similarly, the Synchronization Patterns [5] approach also decouples synchronization from the functional code in a class definition. The approach specifies synchronization constraints using data structures, operations, locks, and synchronization schemes– specified in terms of mutual exclusion, pre and post conditions. From the definition of the synchronization patterns and the 'normal' class definition, code for the target language is generated using a rich meta-language.

Other advocates of concurrency reuse suggested guard-based approaches such as the Threaded Active Object (TAO) [6] model that allows specification of a guard at a method's interface. The guard follows the same principle as the object-oriented paradigm in that it can be overridden and extended at the new method if the base guard needs to be changed.

Another closely related technique to the guard-based approach is the Composition Filter [7] approach. Input and output messages to and from an object go through a set of programmed input and output filters. A filter consists of three parts: a condition, a pattern, and a substitution. Also, the filters may change the semantic of the invoked object. Synchronization constrains are specified using a wait filter – essentially acting as a guard. The filter passes successful messages to the next filter, and the failed ones will be stored in a queue until they become acceptable. The behavior coordination between objects is done by the Abstract Communication Types (ACT).

References

[1] G. Brian, Java Concurrency in Practice, Addison-Wesley, 2006.
[2] AspectJ, http://www.eclipse.org/aspectj/
[3] David Holmes, James Noble, John Potter, "Aspects of Synchronization", Proceedings of the Technology of Object-Oriented Languages and Systems, IEEE Computer Society, pp 2, 1997
[4] Lopes C. V., "D: A Language Framework for Distributed Programming", Ph.D Thesis, College of Computer Science, Northeastern University, November 1997
[5] C.V. Lopez and K.J. Lieberherr., "Abstracting process-to-function relations in concurrent object-oriented applications", Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP), LNCS, 1994
[6] Mitchell S. E., "TAO – A Model for the Integration of Concurrency and Synchronization in Object Oriented Programming", Ph.D dissertation, Dept. of Computer Science, York University, UK, 1995 [4] L. Bergmans, "Composing Concurrent Objects", PhD Thesis, University of Twente, 1994
[7] L. Bergmans, "Composing Concurrent Objects", PhD Thesis, University of Twente, 1994