Journal of Software Engineering

ISSN 1819-4311

Journal of Software Engineering 5 (3): 108-115, 2011
ISSN 1819-4311 / DOI: 10.3923/jse.2011.108.115

A Scheme Based Paradigm for Concurrent Programming

Nathar Shah and Visham Cheerkoot
Faculty of Information Technology, Multimedia University, Jalan Multimedia, 63100 Cyberjaya, Selangor, Malaysia

Corresponding Author: Nathar Shah, Faculty of Information Technology, Multimedia University, Jalan Multimedia, 63100 Cyberjaya, Selangor, Malaysia Tel: +60383125230 Fax: +60383125264

ABSTRACT
Advances in computer architectures, namely the prevalence of muticore architecture have raised challenges for software developers to take advantage of the parallelism. The paper introduces synchronization scheme based concurrent programming that is easy (i.e., GUI based construction), robust and reusable. The synchronization schemes consist of several synchronization units that can be composed together. The schemes can then be applied to code to achieve required concurrency and parallelism. This is done by configuring rather than coding. It is robust because the CScheme engine is linked to a pluggable concurrency bug detection engine. Reusability can be achieved by reusing the scheme on several similar nature problems. We illustrated the architecture of our CScheme engine and discussed the components in it that fulfilled our objectives. Few examples of the synchronization schemes such as Single Threaded Execution Scheme, Reader Writer Scheme and Thread Coordination Scheme together with their units were also build to prove our approach.

Key words: Concurrent programming, object-oriented programming, development environment, programming paradigm, multithreading, scheme based programming

INTRODUCTION
Computer architecture has undergone significant changes in the past 10 years. Now the burden is on the developers to explicitly parallelize their applications in order to take full advantage of the increasing number of cores that each successive multicore generation will provide (Adl-Tabatabai *et al*., 2006). Through this paper, we describe a software paradigm, named CSchema that will relieve an applications developer from the issues of parallelization and synchronization while still utilizing the full power of multicore CPUs that are available today. This tool provides concurrency constructs that help to manage concurrent access to shared memory by multiple threads.

The main contribution of our work is in devising a new paradigm for concurrent programming that hinge on the concept of robustness and reusability. The concept is elaborated with a few practical examples.

We make use of aspect-oriented programming (Kiczales *et al*., 1997) as the main approach to our solution. This technique has proved to increase software modularity in practical situations where object-oriented programming does not offer an adequate support.

Using AspectJ (The AspectJ Team, 2001), a Java based aspect oriented language; all the synchronization concerns of a particular object-oriented program can be separated thereby alleviating the need for a programmer to take care of synchronization issues while developing an object-oriented application. Such synchronization concerns can be configured using AspectJ's aspects. By using the CScheme tool, these synchronization aspects do not need to be developed by the programmer. Instead, we provide Synchronization Schemes which are pre-configured templates that define specific thread coordination and communication mechanisms. Moreover, each Synchronization Scheme comprises of several Synchronization Units that can be further configured to take care of specific synchronization issues. Furthermore, the provided Synchronization units can be composed to build custom developer-defined Synchronization Schemes based on the synchronization needs of the object-oriented software that he/she is developing.

SCHEME BASED CONCURRENT PROGRAMMING

A scheme can be defined as an organization or outline of concepts that is applicable to a general conception. Scheme-based programming is a type of programming that deals with schemes which encapsulate a combination of semantics that can be applied to programs with similar needs. Such needs may include synchronization features and security features amongst others. A Synchronization Scheme can therefore be defined as a grouping and encapsulation of synchronization concerns that can be used for implementing, abstracting and composing synchronization features of an application. As such, programs with similar synchronization needs can use the same scheme.

In this study, we propose a scheme-based concurrent programming that makes use of synchronization schemes. We provide a GUI tool to build such synchronization schemes. The building blocks of our proposed synchronization schemes are called Synchronization Units which encapsulate synchronization as well as thread interaction mechanisms. The pre-defined schemes that we provide come with several synchronization units that can be used to configure different implementations of the synchronization scheme. Moreover, compatible schemes can be composed to build more complex ones, based on the synchronization needs of the programmer. Our GUI tool provides drag-and-drop facility, thereby minimising the coding of an application's synchronization features. The code for the graphically created synchronization schemes is then auto-generated by our engine and threads in the target program will only interact based on the mechanism encapsulated by the applied scheme's configuration. When developing a concurrency application, there are a lot concurrency issues that can arise. These issues can be classified in two general categories: shared data and thread coordination and performance issues.

The issues related to shared data can arise as a result of the following: modification of data by multiple threads without proper locking thereby resulting in data corruption; sharing mutable static variables across threads; synchronizing on a null variable; changing the instance on which we are synchronizing on in one part of the program; synchronizing on string literals and autoboxed values; synchronizing on a re-entrant lock; visibility issues such as protecting writes but not reads thereby resulting in lost updates; improper guarding of non-atomic operations; use of non-atomic 64-bit values; unsafe publication by publishing a reference to "this" in the constructor.

The problems that can arise as a result of improper thread coordination are due to: a call to Thread.stop()-causes all monitors to be unlocked; a call to Thread.suspend() or Thread.resume()-can lead to deadlock; a call to Thread.destroy()-this is unsafe; a call to Thread.run()-this will never start a thread; improper use of wait()/notify().

Performance issues in a multi-threaded program arise mainly because of deadlocks starvation and live-locks.

The Single Threaded Execution Scheme consists of two synchronization units, the Captured Lock synchronization unit as well as the Shared Lock synchronization unit. Using this scheme, issues related to Shared Data access by multiple threads can be solved. For example, they can be used for the proper guarding of an object's shared states against access by multiple threads simultaneously. Issues related to non-atomic operations can also be solved using this scheme. Hence, this scheme can ensure visibility and atomicity of any guarded shared state.

Next, the Reader Writer Scheme can be used to coordinate access to a shared object by reader and writer threads. Reader threads are not allowed to modify any state of a shared object. This solves issues related to data corruption thereby ensuring visibility and atomicity. Moreover, resource starvation can be eliminated by appropriate use of the synchronization units. The Reader Writer Fair synchronization unit always assigns a released lock to the longest waiting threads.

Then, the Guarded Suspension Scheme provides solutions to issues related to liveness issues. Livelocks due to spin-locks can be prevented using this scheme. Using this scheme, preconditions can be specified upon which a thread can wait on if the requested resource is not available. Moreover, once this shared resource is released, the appropriate waiting threads will be successfully notified.

Lastly, the Threads Coordination Scheme provides mechanisms for coordinating threads communication. It provides simpler ways, as compared to other synchronization constructs such as wait()/notify(), for thread communication. As a result, the risks of deadlocks and livelocks are minimized since issues related to wait() and notify() need not be taken care of. There are two synchronization units, the Count Down Latch and the Cyclic Barrier. The former is useful when a one-time thread coordination is required such as starting a set of threads together to achieve a particular task. The CyclicBarrier is generally more useful than CountDownLatch in cases where a multithreaded operation occurs in stages or iterations and a single-threaded operation is required between stages/iterations, for example, to combine the results of the previous multithreaded portion.

The third-party open-source bug detection engine, CheckThread (Joe, 2009) that we are using makes use of static analysis to find concurrency bugs at compile time. It can detect whenever an unsafe JDK class is being used in a concurrent program. It can also detect race conditions whenever non-atomic operations are performed. Simple cases of deadlock situations can also be reported. It also detects issues related to threads coordination highlighted earlier in this section.

DESIGN OF PROPOSED PARADIGM

Cscheme tool provides the ability to incrementally build Synchronization Schemes which encapsulate synchronization constructs and thread interaction mechanisms. The ultimate goal is to separate concurrency concerns from the normal coding of resource classes by defining and configuring such concerns with minimal extra lines of code. We also provide a graphical user interface that greatly helps to ease the process scheme building and configuration through features such as drag-and-drop.

An overview of the design of our CScheme system is provided in the component diagram overleaf.

An application usually consists of resource classes, utility classes and driver classes amongst others. This is depicted by component A on the component diagram. Usually, the resource classes encapsulate several states, some of which need to be updated atomically for consistency and to prevent lost updates. Therefore, for a concurrent application, the objects of resource classes need to be properly guarded from simultaneous access by multiple threads. Any failure to ensure proper synchronization during multi-threaded access to a shared object will result in havoc during program execution. We therefore provide a simple way to mark resource class attributes that can be the subject of concurrent modification. We provide an annotation (@SharedField, shown in component B) to mark such states. This is done manually by the developer.

In order to detect all the shared fields, we make use of static analysis (Ayewah *et al.*, 2008) to analyze the source code as well as the byte-code of the resource classes. The static analysis libraries that we make use of are the Byte Code Engineering Library (BCEL) (Dahm 2001) and Chiba (1998). Through static analysis, we can easily determine whenever an attribute, tagged with the "@SharedField" annotation, will be modified in a method.

Figure 1 presented CScheme component diagram. Component G, the resource class analyzer, implements these functionalities by making use of the afore-mentioned libraries. This component is also used to mark the methods that make use of the shared fields. The usage of these marked methods is explained later in the paragraphs below.

Synchronization Schemes creation and configuration are done by making use of the GUI (Fig. 2) that comes with our tool. Component D gives an overview of this process. The drag and drop features provided by the GUI aids the programmer to build a scheme by selecting the appropriate synchronization unit and configuring it with the required information based on his synchronization needs. Some schemes along with their associated synchronization units are shown in component C.

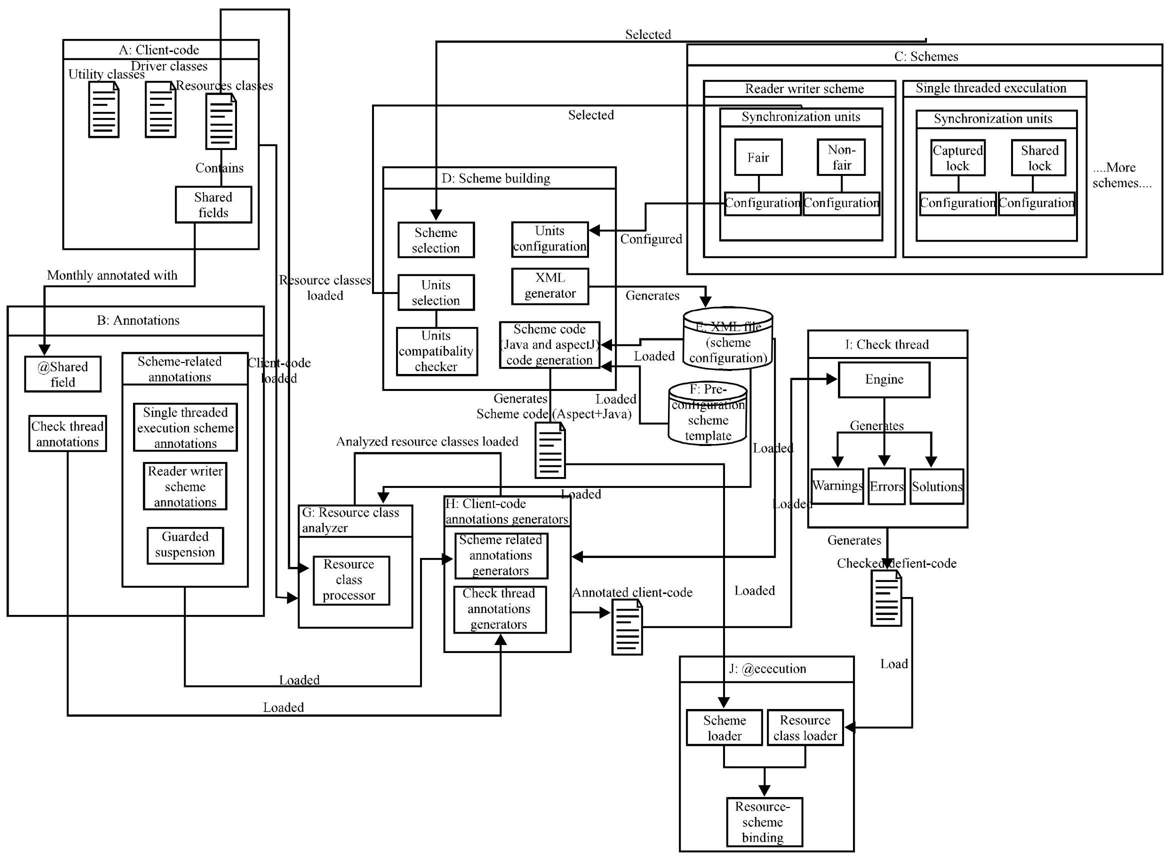
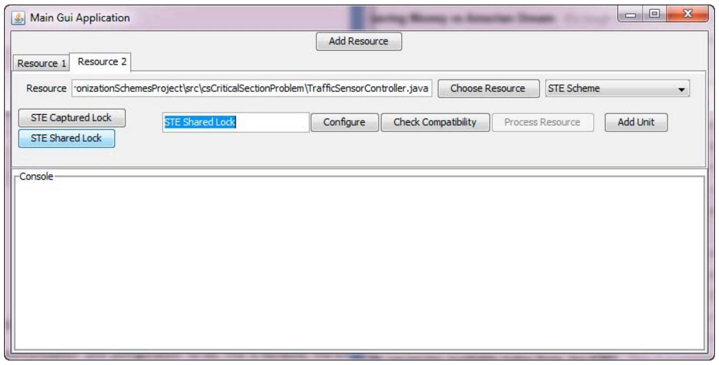


Fig. 1: Cscheme component diagram



Resource: ·onizationSchemesProject\src\csCriticalSectionProblem\TrafficSensorController.java

STE Scheme

STE Shared Lock

Fig. 2: The GUI interface of CScheme

All scheme configurations are stored in an XML file (component E) so that the scheme building component can make use of these configuration data along with the pre-configured scheme templates (component F) to generate the desired synchronization scheme as AspectJ and Java code (output of component D). Some of the schemes that we provide, the Guarded Suspension Scheme and the Thread Coordination Scheme, have preconfigured templates thatis they have already been pre-populated with synchronization and thread coordination mechanisms that can be applied to any class that will make use of such schemes. The remaining configuration of these schemes is determined from component E (user-defined configurations in the XML file).

Once, the schemes that the developer created graphically has been generated as their respective AspectJ and Java codes, they need to be applied to the target resource classes. In order to determine which parts of the resource class that will be affected by the applied scheme, the resource class needs to be marked using annotations to specify those parts. We make use of component G, the resource class analyzer, to determine methods in which shared fields may be subject to concurrent access by threads. Once, these methods have been determined, we can make use of component H to generate the necessary annotations based on the configured schemes. However, such methods (and the resource class as well) may still contain concurrency bugs. Therefore, we provide a reliable bug detection mechanism by making use of a third-party open-source tool called CheckThread (Joe, 2009), shown as component I. The latter makes use of static analysis as well to catch concurrency bugs, such as detection of race conditions or the use of unsafe collections, at compile time. It also provides solutions to the most common concurrency bugs if they are detected in the client-code. The CheckThread annotations are auto-generated in the component H using the static analysis done in component G.

Once all the required annotations have been generated in the resource class, the latter will be weaved with the generated scheme codes by the AspectJ weaver during execution and consequently any multi-threaded access to resource objects and their shared states will be dictated by the applied synchronization scheme.

Schemes and synchronization units description

Single threaded execution scheme: Some methods access data or other shared resources in a way that produces incorrect results if concurrent calls to a method accessing the data or another resource at the same time. The Single Threaded Execution scheme solves this problem by preventing concurrent calls to the method from resulting in concurrent executions of the method.

When to use this scheme: A class has methods that update or set instance or class variables; a method manipulates external resources that support only one operation at a time; the class's methods may be called concurrently by different threads.

There are 2 Synchronization Units associated with the Single Threaded Execution Scheme. STE Shared Lock-uses the monitor of the target object for synchronization. STE Captured Lock-uses the scheme's aspect monitor to control the access to all captured joinpoints/methods that match the pattern defined for STE Captured Lock. Moreover, this scheme allows the user to select any specific method from the Resource class to force it to follow the synchronization pattern encapsulated by the STE Captured Lock.

Annotations generated by single threaded execution scheme and its synchronization units

Class-level annotations: @SingleThreadedSchemeCapturedLockManaged-for a Resource class that has been configured with the STE Captured Lock Synchronization Unit. And @SingleThreadedSchemeSharedLockManaged-for a Resource class that has been configured with the STE Shared Lock Synchronization Unit.

Method-level annotations: @CapturedLock-for methods in a Resource class that has been configured with the STE Captured Lock Synchronization Unit. These methods have been identified by the Resource Class Analyzer component of the engine.@SharedLock-for methods in a Resource class that has been configured with the STE Shared Lock Synchronization Unit. These methods have been identified by the Resource Class Analyzer component of the engine.

Reader writer scheme: This scheme allows concurrent read access to an object but require exclusive access for write operations. It coordinates concurrent calls to methods that read or change an object's shared, mutable state.

When to use this scheme: There is a need for read and write access to an object's state information; any number of read operations may be performed on the object's state information concurrently. However, read operations are guaranteed to return the correct value only if there are no write operations executing at the same time as a read operation; write operations on the object's state information must be performed one at a time, to ensure their correctness; there will be concurrently initiated read and write operations; allowing concurrently initiated read operations to execute concurrently will improve responsiveness and throughput.

There are 2 Synchronization Units associated with the Reader Writer Scheme: RW Fair and RW Non-Fair.

RW Fair-this Synchronization Unit is based on the implementation of the java.util.concurrent. locks.reentrantreadwritelock. This Synchronization Unit ensures that threads contend for entry using an approximately arrival-order policy. When the currently held lock (encapsulated by the Synchronization Unit) is released either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads that group will be assigned the read lock. RW Non-Fair-when this Synchronization Unit is used, the order of entry to the Resource class that is bonded by this scheme is unspecified, subject to reentrancy constraints.

Annotations generated by single reader writer scheme and its synchronization units
Class-level annotations: @ReaderWriterFairSchemeManaged– for a Resource class that has been configured with the RW Fair Synchronization Unit. @Reader Writer Non-Fair Scheme Managed -for a Resource class that has been configured with the RW Non-Fair Synchronization Unit.

Method-level annotations: @Write only-for methods in a resource class that have been identified by the resource class analyzer component of the engine that write to and/or read fields that have been annotated with the @sharedfield annotation and @readonly -for methods in a Resource class thathave been identified by the resource class analyzer component of the engine, that only read fields that have been annotated with the @SharedField annotation.

Thread coordination scheme: This synchronization scheme provides thread coordination mechanisms that can be used in situations when: (a) there is a need to start several threads at the same time; (b) there is a need to wait for several threads to finish; (c) a multithreaded operation proceeds in multiple stages; (d) a single-threaded operation is required between stages of a multithreaded operation.

This synchronization scheme provides two synchronization units to achieve the above-mentioned requirements. The first synchronization unit under this synchronization scheme is a Count Down Latch and it can be used in situations (a) and (b). The second synchronization unit is a Cyclic Barrier and it caters for situations (c) and (d).

For this synchronization scheme, client code need not be annotated as in this case, classes are generated and the developer can code the multithreaded logic of his application in methods provided by these generated classes. Traditionally, it has been mentioned that concurrency is a crosscutting concern that tangles in application code (Kiczales *et al.*, 1997).

CONCLUSION

This study is significant to innovatively apply the separation technique to build a new programming paradigm, named CScheme for concurrent/parallel programming. The work is important as it device a new paradigm for productive concurrent programming based on software engineering attributes such as robustness, reusability and configurability. The perils of concurrent/ parallel programming can be left to the expert concurrency programmers to develop the synchronization scheme and ordinary programmers can reuse the schemes to take advantage of the parallelism required in the prevalent parallel architectures in the consumer machines.

REFERENCES

Adl-Tabatabai, A.R., C. Kozyrakis and B. Saha, 2006. Unlocking concurrency. Queue-Comput. Archit., 4: 25-33.

Ayewah, N., D. Hovemeyer, J.D. Morgenthaler, J. Penix and W. Pugh, 2008. Using static analysis to find bugs. IEEE Software, 25: 22-29.

Chiba, S., 1998. Javassist: A reflection-based programming wizard for Java. Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C^{++} and Java, October 1998, International Business Machines Corporation, Japan.

Dahm, M., 2001. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universitat Berlin, Institut fur Informatik, Germany.

Joe, C., 2009. Cool concurrency with CheckThread. eclipse CON™. http://eclipsecon.org/2009/sessions?id=307

Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin, 1997. Aspect-oriented programming. Proceedings of the European Conference on Object-Oriented Programming, 1997, New York, pp: 220-242.

The AspectJ Team, 2001. The AspectJ™ programming guide. Xerox Corporation, 2002-2003, Palo Alto Research Center, USA. www.eclipse.org/aspectj/doc/released/progguide/index.html