# A synchronized block join point for AspectJ

Chenchen Xi Bruno Harbulot John R. Gurd

The University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom

xic AT cs.man.ac.uk bruno.harbulot AT manchester.ac.uk jgurd AT cs.man.ac.uk

## Abstract

Designing and implement model for a synchronized block join point to encapsulate crosscutting synchronization concerns into single unit in AspectJ.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Design, Language.

***Keywords*** Synchronized Block Join Point, AspectJ.

## 1. Introduction

Synchronization is a concern that developers of distributed systems must deal with whenever guarded access to a shared resource is required. Access to devices, files and shared memory are all situations that typically require synchronization, and they also often require careful management of multiple threads and synchronization devices such as locks. Avoiding tangling of the code responsible for these various concerns is difficult, as is encapsulating them for reuse in diverse situations.

In the context of distributed Java programs, three distinct synchronization situations are encountered, namely, sharing data among multiple threads, sharing among clusters of JVMs and sharing among clusters of physical computers. The former is the logical concern of sharing data between concurrent threads; the latter two are practical concerns about the particular 'distributed environment' in which the code is executed. The problem is how to distribute the logically necessary Java threads transparently across the physical vagiaries of the distributed environment.

Moreover, parallel programming is often difficult simply due to the complexity of dealing with lock-based synchronization. As a result, there have been proposals to simplify parallel programming by using various forms of *transactional memory* to replace lock-based synchronization in existing parallel Java programs. Once *Java Memory Model* (*JMM*) issues have been addressed, conversion of lock-based synchronization into transactions is largely straightforward. However, it is still problematic to avoid code tangling while effecting this conversation.

*Aspect-Oriented Programming* (*AOP*) has the potential to modularise such synchronizations so that user code can become oblivious to the distributed environment. Indeed, a join point based on the `synchronized` method has been proposed. However, the `synchronized block` has not yet been treated in *AspectJ*.

This paper [1] shows how to separate the synchronization concern by designing and implementing models for a synchronized block join point and a synchronized block body join point to encapsulate crosscutting synchronization concerns into logical units using the concepts of *AOP*. It is also shown how the two new join points can help such modularization to plug into the *JMM* so as to maintain its semantics, along with the semantics defined in the *Java Language Specification*. The models achieve reusability of synchronized code and thread control management in Java to such an extent that concurrency can be fully handled by a single aspect. The models augment the capabilities of join points for synchronized methods in intercepting and modifying synchronization actions in distributed, Java-based, aspect oriented software. The work is applicable in any aspect oriented environment, but emphasis is placed on compatibility with the most commonly used language *AspectJ*.

The proposed join point model is enhanced with a mechanism for removal of unnecessary synchronization, which is vital for reducing overheads associated with the lock. There is also a facility for re-introducing necessary synchronization that has previously been removed.

```
void around(HashMap map): synchronize()
          withincode(method()) && args(map){
          atomic{ rm_proceed(map) ; }  }
```

Consider the above simple example, which is captured by the `synchronize()` pointcut. With transactional execution, there is no need to use anything other than a non-locking HashMap since the caller specifies its atomicity requirements. The `rm_proceed()` method provides the ability to remove the synchronization on HashMap `map` and `atomic` warp it as a transactional object. Extensions for the `abc` compiler which implement the two new join points are briefly presented and are shown to meet the requirements of various applications.



*Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008),* April 1, 2008, Brussels, Belgium.



[1] Full text can be found at http://www.cs.man.ac.uk/~xic/SBJP_AspectJ.pdf