Synchronizing Java Threads Using Assertions

Miguel Katrib, Damián Fernández
Dpto. de Ciencia de la Computación, University of Havana, Cuba
mkm@matcom.uh.cu

Ernesto Pimentel
Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga, Spain
Ernesto@lcc.uma.es

Abstract

The existence of mechanisms to describe and evaluate assertions in programming languages helps to reduce the conceptual gap existing between the specification and implementation phases in the software development, improving both tasks. Java, originally used as an Internet-oriented implementation language, is becoming a general-purpose language, which unfortunately does not provide this kind of mechanisms. In this paper, we present a proposal to include assertions into Java, which allows also including the use of quantified variables in the assertions. The paper analyzes the properties that such integration must satisfy in order to guarantee a seamless interaction with the rest of the language constructions. The proposal shows how assertions can be used to synchronize Java threads introducing the notion of synchronization object and extending the semantics of preconditions. The goal is to increase the object oriented expressiveness of the language to describe concurrency.

Keywords: Concurrency, Synchronization, Object-Oriented, Assertions, Java

1. Introduction

Assertions are logic clauses used in program specifications during the analysis and design phases of software development. At the programming and verification phase assertions are useful for debugging and verification. Assertions can be integrated into the object-oriented model via the metaphor of Design by Contract [9,10]. This metaphor presents two main advantages. Analysis, design and implementation are improved because the assertions are a nexus between the different development phases, and because they make the modeling and the documentation easier. On the other hand, they could improve issues like reliability and security, since they can be monitored in runtime.

Unfortunately, Java lacks mechanisms to apply this design strategy. The role of pre and post-conditions in Java has already been considered by Papurt [14], but only from a methodological point of view. [6] argues about the importance to include assertions in Java and makes a review of different approaches to do this task (see also the discussion list about this matter initiated by Mike Mannion in http://developer.javasoft.com/developer/bugParade, bug 4071460.). JaWa (Java with assertions) is an extension developed in the University of Oldenburg [8]. Major flaws of JaWa are the exclusion of an Eiffel-like old operator in postconditions and the insufficient integration between assertions and inheritance. In JaWa if a method with assertions is overridden in a subclass, then the redefined method must also include its own assertions; otherwise a compile time error is generated. AssertMate [15] is another Java extension based in the Design by Contract metaphor. AssertMate, as our proposal, also allows the use of quantifiers in the assertions. Nevertheless AssertMate does
not give any treatment related to assertions and inheritance. A number of attempts to include assertions have also been done for others languages, see, for example, in [1] how assertions were included in Smalltalk. The present paper summarizes how Java can be extended with assertions (for brevity we will call this extension JavaA) and analyzes how assertions can be integrated with Java threads for concurrency.

Several proposals about the integration of object-orientation and concurrency has been studied [13]. One of the most widely studied approaches has been the usage of synchronization guards in the methods of a class. This idea has been already used in a concurrent version proposed for Eiffel [11,12].

Thus, the inclusion of assertions in Java could also provide an alternative to the way in which the synchronization is made in that language. Therefore, this paper proposes a double semantic for the evaluation of preconditions, avoiding in this way the explicit use of monitors and semaphores, such as it happens in Java. This will unburden the programmer from the description of non-relevant synchronization details.

The aim of this work is to propose an integration of assertions (à la Eiffel) in the Java language and to study how to use those assertions in describing concurrency issues. The paper is organized as follows. In the next section, we give the premises for the inclusion of assertions. In Section 3, we describe the way in which the assertions have been integrated into Java. Section 4 discusses the concurrency mechanisms of Java, comparing them with those that are present in Eiffel, and introduces the notion of synchronization object and explains how to use preconditions when they are combined with synchronization objects as a more object oriented alternative to synchronize Java threads. Section 5 shows some implementation details. Finally, we give some conclusions.

2. Premises of the integration

The impedance between the underlying language (Java) and the extended language (JavaA) is one of the inconveniences we face when extending a programming language. The proposed extension in JavaA must satisfy the following premises.

1. The syntax of JavaA has to be similar to that of Java. To be non-invasive, assertions are included as special Java comments, so that a text in JavaA can be compiled as a traditional Java program by a Java compiler.

2. A "Preprocessing Tool" to translate a JavaA source class to an equivalent Java target class (and then to the corresponding .class file) must be constructed.

3. A Java client class should have the choice of using a JavaA .class file without knowledge about JavaA (even in absence of the Preprocessing Tool) but enjoying the benefits provided by the embedding of the assertions in the .class file. These benefits are:
   - Preconditions prevent errors in the design and implementation of the client class.
   - Postconditions and invariants protect the client class against errors in the supplier class (the class originally coded in JavaA).

4. The Preprocessing Tool (2) only has to be used when the designer of the client class wants to include his/her own assertions, or when a subclass extends a class written in JavaA and wants to benefit from the presence of assertions in its superclass.

5. In order to support documentation a "reflection" tool (JAI – Java Assertion Interface –) must be provided to extract the “interface” of a JavaA class (i.e. assertions plus method’s signatures) from the .class file.

6. The integration between the assertion mechanism and the Java concurrency mechanism should be considered.
Throughout the present paper we introduce several "rules" which must be guaranteed by the Preprocessing Tool implementing the assertion mechanism, or which must be granted as a methodological recommendation by the developer.

3. Eiffel-like Assertions in Java

In this section we summarize the syntax and the basic semantics of the inclusion of assertions in JavaA. For more details and for the approach used in the implementation see [4].

3.1. Preconditions

A precondition of a method $f$ is written in JavaA as

$$f(...)
/**require {<Assertions>}**/
...body of the method
$$

It is located between the header and the body of a method. $<Assertions>$ is a sequence of assertions separated by semicolons, i.e. $<assertion> ; ... ; <assertion>$. Each assertion is a Java boolean expression (maybe with some especial constructions in the case of postconditions) or a quantified predicate (that will be explained below). To execute the body of the method, all the assertions must evaluate true.

If the method $f$ is a redefinition, then the precondition must be introduced by a require else clause,

$$f(...)
/**require else {<Assertions>}**/
...body of the method
$$

In this case, the evaluation of the precondition is $<Assertions>$ or else $<Assertions$ in the overridden method$>

This guarantees that the precondition of the redefined method is weaker than that of the parent. The Design by Contract metaphor is based on the fact that, before invoking a method $f$ of that class, the client of a class $C$ must guarantee that the precondition of $f$ is fulfilled. Hence, the designer of the class $C$ must be careful to ensure that methods or variables used in a precondition of a method $f$ are visible to the same clients to which $f$ is visible.

3.2. Postconditions

The postcondition of a method is written after its body,

$$f(...)
...body of the method
/**ensure {<Assertions>}**/ }$$

The postcondition has to be true after the normal execution of the body. Two special constructions can also appear in the Java boolean expressions used in postconditions: the result special variable and the old operator. Both constructions have a similar meaning to those defined in Eiffel. If the method $f$ overrides a parent method, then the postcondition must be introduced by an ensure then clause,

$$f(...)
...body of the method
/**ensure then{<Assertions>}**/ }$$

In this case, the evaluation of the postcondition is $<Assertions>$ and then $<Assertions$ in the overridden method$>

According to the Design by Contract metaphor this guarantees that postcondition of the redefined method is stronger than precondition of the parent.
3.3. Invariants

An invariant has the form

```java
class A{.../**invariant {<Assertions>}**/}
```

It is located before the closed curly brace "}" at the end of the class. The invariant of a class must be successfully evaluated in every stable state of the object, i.e. after and before the evaluation of any visible method of the class. Invariants are preserved by inheritance; i.e. the invariant of all the superclasses in the inheritance path of the class must be also satisfied each time an instance of a subclass is used.

3.4. Quantifiers and assertions

Some proposals to include quantified assertions in Eiffel can be found in [2,3,6,16]. AssertMate [15] also introduces quantifiers in Java. JavaA provides a universal quantifier, `forall`, and an existential quantifier, `exist`.

A problem that appears when a quantifier is used in an actual implementation language (such as Java) is to define the "domain" on which the quantification is applied. The approach used in JavaA is supported by the `Enumeration` interface. The assertion `forall x in c: (<Boolean expression>)`, where `c` is an instance of a class implementing `Enumeration`, will evaluate true if for all objects returned by `c`, the boolean expression evaluates true.

Analogously, the existential quantifier can be written as

```java
exist x in c: (<Boolean expression>)
```

is true if the `<boolean expression>` is true for at least one element in `c`. As Java does not support genericity, then the method `nextElement()` of the interface `Enumeration` returns a value of the type `Object`. Thus, in JavaA the quantified variable is implicitly typed `Object`. Therefore, the programmer must cast this variable into the correct type.

Examples about the usefulness of the quantifiers to increase the expressiveness of the assertions can be found in [3,4].

3.5. Assertion violation

Assertions in JavaA are monitored in runtime. To keep JavaA as closed as possible to Java, no special constructions to handle assertion's violation (like the rescue clause existing in Eiffel), were introduced. If an assertion evaluates false an exception will be raised. This exception must be caught in the traditional `try-catch` Java syntax.

4. Using assertions to synchronize JavaA threads

Java provides a concurrence mechanism based on threads, which is enabled by inheriting from the class `Thread` or by implementing the interface `Runnable`. To protect the consistency of objects used in different threads it is necessary to provide a mechanism to describe critical sections. This is achieved in Java by the qualifier `synchronized`. Access to a method, or a piece of Java code, qualified in this way, is controlled by the object's monitor. To hand over the control from one thread to others, operations `wait()` or `notify()` should be used explicitly. This approach down loads the burden of the design to programmers (more error-prone) and emphasizes the responsibility of synchronization on the code and not on the objects. Other more natural proposals in the context of object-oriented languages [13] offer mechanisms with a higher level of abstraction. This is also the case of Eiffel [2,3], where synchronization is based on preconditions, providing a double semantic for the evaluation of a precondition. The dual behavior associated with preconditions, together with the possibility of defining separate objects, makes the Eiffel approach to model concurrent problems, friendlier
and less error-prone than those of Java. Nevertheless, the Eiffel intention of introducing only one linguistic resource (separate object) forces the programmer to fulfill a set of rather restrictive rules, which implies in some cases, unnatural designs from an object perspective.

The present proposal tries to achieve object-oriented concurrent programming in Java using preconditions to synchronize threads. The technique is based on the notion of synchronization object. It combines some of the basic ideas of the concurrency models of Java (threads) and preconditions to synchronize, thus producing more natural object-oriented designs, making the programmer job easier, and potentially avoiding deadlock situations. Then, the Java programmer has only to define what the synchronization objects are, and to design the synchronization by using those objects in preconditions.

4.1. Synchronization Objects and Threads

An object serves as a synchronization object if its class implements the interface Synchronizable. The presence of a synchronization object in the precondition of a method $f$ means that the precondition will also be used for thread synchronization.

**Rule 1**

| The presence of a synchronization object in the precondition of a method $f$ will be considered only when it is in a static way. That is, through an instance variable, a function call or a formal parameter of $f$ that must be declared of static type Synchronizable. |

If one of the boolean expressions in the assertions evaluates false, and it does not include synchronization objects, then the evaluation of the precondition will be considered in the usual way: a violation of the contract between the method and its client and, therefore, an exception will be raised. On the contrary, if only those boolean expressions including synchronization objects evaluates to false, then the new semantics for the precondition evaluation will be engaged: the suspension of the thread running the caller method. In this case, synchronization objects participating in the precondition will not be locked, allowing them to be used by other threads.

Because of a precondition can be expressed through several boolean expressions separated by semicolons, then the developer has the opportunity to tune the precondition distinguishing those used for synchronization from those used as in the traditional sequential way. To evaluate a precondition having synchronization objects, they must not "be in use" (locked) by others threads. When a method $f$ needs synchronization objects (it uses them in its precondition) it must "lock" them to evaluate the precondition. If the precondition is satisfied, the method (the thread running the method) keep the synchronization objects locked until finish its execution. Whereas the method's body is executing, no other thread will have access to these objects. When the execution of the method finishes, the locked objects are unlocked. It’s worth noting that in this case other suspended preconditions may succeed (either because the synchronization objects are now available or because they changed their state).

To syntactically guarantee the presence of a synchronization object in simple preconditions (those that don't need any other property from the synchronization object) of a method $f$, the interface Synchronizable includes the boolean method available(). If the precondition of a method is not satisfied then, to avoid deadlock and unrecoverable situations, every (not previously locked by the thread) synchronization object will be released (remain unlocked) and the execution of the method will be suspended.

**Rule 2**

| To avoid unexpected effects, a method can only change the state of a synchronization object if this object is used in its precondition. |
This rule is less restrictive and more object-oriented than the one proposed by Meyer for Eiffel [3], where “separate objects” used to synchronize a method of another separate object can only be accessed if they are used as formal parameters of the method.

In a chain of calls (running under the same thread) only the first method, which has locked a synchronization object (using it in its precondition), can unlock it.

Rule 3

The first method in a chain of calls which locked a synchronization object, must unlock the object before exiting.

4.2. The Consumer-Producer Example

Let’s consider two classes, Producer and Consumer. Instances of these classes may produce and consume items, respectively. They synchronize by using a shared data structure (a synchronization object of a type Buffer):

```java
class Buffer extends Monitor {
    ... 
    boolean full();
    boolean empty();
    void put();
    void remove();
    Object item();
}
```

The class Monitor is an implementation of the interface Synchronizable. Thus, the class Producer could be defined as follows.

```java
class Producer implements Runnable {
    Buffer queue; //It shares the queue with the Consumer
    Producer(Buffer q) {
        queue = q;
        new Thread(this, "producer").start();
    }
    void store(T y) {
        queue.put(y);
    }
    void run() {
        T x;
        while (some condition) {
            //...to produce an item x of type T
            store(x);
        }
    }
}
```

```java
class Consumer implements Runnable {
    Buffer queue; //It shares the queue with the Producer
    Consumer(Buffer q) {
        queue = q;
        new Thread(this, "consumer").start();
    }
    T get() {
        T y = queue.item(); queue.remove(); return y;
    }
    void run() {
        while (some condition) {
            T x = get();
            //...to consume an item x of type T
        }
    }
}
```
The preconditions in the methods `store()` and `get()` use the synchronization object `queue` by sending the query messages `full()` and `empty()`, respectively.

### 4.3. The Eating Philosophers Example

The following example illustrates the current proposal by means of the well-known philosopher’s problem.

```java
class Fork extends Monitor{ }
class Philosopher extends Thread {  
    Fork left;  
    Fork right;  
    Philosopher(Fork l, r){ left = l; right = r;};  
    void run { while (.some condition. )  
    { think(); eat();}  
    void think()  
    {...};  
    void eat()  
    /**require {left.available(); right.available()}**/  
    { //...eating}  
    }  }  
Fork[] forks = new Fork[4];  
new Philosopher(forks[0], forks[1]).start();  
new Philosopher(forks[1], forks[2]).start();  
new Philosopher(forks[2], forks[3]).start();  
new Philosopher(forks[3], forks[4]).start();  
new Philosopher(forks[4], forks[0]).start();
```

Note how this solution avoids deadlock. The method `eat()` acquires both forks by means of its precondition `/**require {left.available(); right.available()}**/` or neither of them. This result is more difficult to achieve using only the Java synchronization resources. Furthermore, the solution above is more natural and object oriented than the one proposed for Eiffel, in which the forks `left` and `right` have to be passed as parameters of the method `eat()`, producing a less object oriented code.

### 4.4. Avoiding potential deadlocks

Under the above explained semantic, a method either acquire all its synchronization objects locking them, or does not acquire any of them. This policy avoids potential deadlock, a thread can not have locked an object and at the same time is waiting for another object locked by another thread. This could easily happen in a bad implementation of the philosopher's example using traditional Java resources.

Another case of deadlock to be avoided is the following one. Consider the following sequence of calls

| Synchronizable | f (Synchronizable x){  
|----------------|------------------------  
| s;             | /**require x.someProp() */  
| ...            | z.h(x)... }  
| a.f(s)         |  

| h(Synchronizable y){  
|-------------------|------------------------  
| /**require y.otherProp() */  
| ... }  

Here `f` is the first method in the sequence of calls that use `s` for synchronization. What should be done if `y.otherProp()` evaluate false? We can not release `y` (i.e. `s`) because `f` is using it. Therefore, in this case to suspend the `h` execution while waiting for some change in `y` will produce a deadlock, because no other thread will acquire and change `s`. In such a
situation the unfulfillment of the precondition \texttt{y.otherProp()} must be considered as a violation of the \texttt{h} contract and then an exception will be raised.

\textbf{Rule 4}

A synchronization object used in the precondition of a method \texttt{f}, running in some thread, will be considered for synchronization purposes only if it is no locked or if it is locked by a different thread. In other cases, the precondition will be considered according to the traditional Design by Contract metaphor.

Observe how the implementation below (method \texttt{someIsFree} in section 5.2) tries to enforce this rule partially.

\section*{5. Implementation}

In order to facilitate the job of the JavaA programmer, an equivalent Java code for the above features must be generated by the Preprocessing Tool. Thus, the programmer should only take care of which the synchronization objects are, which threads he/she needs, and which synchronization protocols will be adequate by means of the preconditions.

Due to space limitations, this paper does not explain how the full assertion mechanism was implemented (for more details see [4]), but we only focus on some aspects related to concurrency and synchronization.

\subsection*{5.1. Synchronization implementation}

A class \texttt{Monitor} does a straightforward implementation of the interface \texttt{Synchronizable}. Java does not provide multiple inheritance, and then programmers must extend \texttt{Monitor} or implement \texttt{Synchronizable} by using a \texttt{Monitor}-like pattern.

When more than one synchronization object is involved in the evaluation of a precondition, it is necessary to guarantee that all these objects will be locked at the same time. This is the only way to avoid deadlock situations, derived from the simultaneous locking of different objects (used in a same precondition) by different threads. Unfortunately, Java does not allow expressions like

\begin{verbatim}
synchronized(x,y){ ...code... }
\end{verbatim}

To solve this kind of situation, we define a class \texttt{Synchronizer}.

\begin{verbatim}
class Synchronizer {
    public static Object sync = new Object;
    public static final boolean lock(Synchronizable[] os){...}
    public static boolean someIsFree(Synchronizable[] os){...}
    public static final void unlock(Synchronizable[] os){...}
}
\end{verbatim}

The \texttt{lock} and \texttt{unlock} methods of \texttt{Synchronizable} must be used only through the synchronized and static methods of \texttt{Synchronizer}. Unfortunately, a selective Eiffel-like export pattern between the interface \texttt{Synchronizable} and the class \texttt{Synchronizer} is syntactically not possible in Java. Then, the developer must conform to the following rule.

\textbf{Rule 5}

The method \texttt{available()} of the \texttt{Synchronizable} interface is the only one which can be directly used by the developer. The other methods must be used only through the \texttt{Synchronizer} class.
5.2. Precondition evaluation

To illustrate how the preconditions are evaluated in JavaA, and can be used to synchronize objects, we will outline the resulting code in Java. Let a method \( f \) in a class \( C \) defined as

\[
\begin{align*}
f() \\
/\*\* require \{ \\
\text{<boolean expr}_1 \text{ not using synchronization objects >; } \\
\text{... <boolean expr}_k \text{ not using synchronization objects >; } \\
\text{<boolean expr}_{k+1} \text{ using synch objects } s_{k+1,1}, s_{k+1,2}, \ldots, s_{k+1,n} >; \\
\text{... <boolean expr}_p \text{ using synch objects } s_{p,1}, s_{p,2}, \ldots, s_{p,n} >; \\
\} \*/ \\
\}
//... body of f
\]

The JavaA Preprocessing Tool will generate the additional method \( f_C\_require \) in the class \( C \) and the instance variable \( f\_os \) of type \( \text{Synchronizable[]} \) if the precondition of \( f \) has synchronization objects.

```java
boolean f_C_require(same parameters as \( f \))
throws PreconditionException {
    f_os =new Synchronizable[]\{s_{k+1,1}, \ldots, s_{k+1,n}, \ldots, s_{p,1}, \ldots, s_{p,m}\};
    while(true){
        if (!<expr>_1) return false; // no synchronizing
        ...
        else if (!<expr>_k) return false; // no synchronizing
        else synchronized(Synchronizer.sync){
            if (!Synchronizer.lock(f_os)) {
                //the method cannot lock all needed synchronizable objects
                if (Synchronizer.someIsFree(f_os))
                    Synchronizer.sync.wait();
                else return false; //due to rule 4 above
            }
            else if (!(<expr>_{k+1} && \ldots &&<expr>_p)) {
                Synchronizer.unlock(f_os);
                if (Synchronizer.someIsFree(f_os))
                    Synchronizer.sync.wait();
                else return false; //due to rule 4 above
            }
            else return true;
            //all synchronization objects are available and
            //all boolean expressions are satisfied
        }
    }
}
```

Then, the code for the body of method \( f \) must include the protocol

```java
f(..parameters..) throws PreconditionException,...{
    boolean preconditionViolation = false;
    try{ if f_C_require(..parameters..){
        //...body of the method f
    }
    else preconditionViolation = true; }
    finally( if (preconditionViolation)
        throw new PreconditionException();
    else synchronized(Synchronizer.sync){
        Synchronizer.unlock(f_os);
        Synchronizer.sync.notifyall(); } }
```

6. Conclusions and further work

Some basic ideas to incorporate assertions into Java were summarized. It was analyzed how Java threads can synchronize by using assertions. The current proposal combines the
thread-based mechanism of Java and the expressiveness of the assertions. The notion of synchronization object and its usage in preconditions was introduced. The integration of threads, assertions and synchronization objects can improve the Java concurrent approach by offering a more object-oriented model and avoiding the use of low level mechanisms as synchronized sections and the wait and notify methods.

The discussed approach gets more natural designs from an object-oriented perspective than the one proposed to Eiffel. In such a proposal for Eiffel the synchronization is restricted to objects passed as parameters to the method. This requires the passing of instance variables as parameters to methods of the same instance, instead the direct use of them in the precondition. We also have presented a number of methodological rules for the correct behavior of the proposed model. A Preprocessing Tool, currently under development, should guarantee them.

On the other hand, the Java model for expressing concurrency is rather poor. In particular, a Java thread object has only a single asynchronous method: \texttt{run()}. This means that a class extending \texttt{Thread} can only express a fixed concurrent behavior. A more object oriented approach, which uses the notion of \texttt{concurrent} (an object whose methods run concurrently with the caller) is now being developed by the authors. Each \texttt{concurrent} object will be attached to a \texttt{logical processor}. Using the Java reflection features this \texttt{processor} will allocate the calls in a queue and will execute the non-query calls asynchronously to the caller method. Two implementations of \texttt{processor} are under development: one using Java threads and another using Java remote execution.

7. Acknowledgments

The authors would like to thank F. Ávila, D. Bello and V. Álvarez from the Object-Oriented Programming group at the University of Havana for their comments and suggestions. This work has been partially supported by the CYTED project IDEAS.

8. References