On synchronization of threads in the Java language using SynchExpressions library

Łukasz Fryz, and Leszek Kotulska,

Institute of Computer Science Jagiellonian University, ul. Nawojki 11, 30-072 Cracow, Poland

Summary
We present a library implementing a new general-purpose synchronization mechanism for Java threads. The mechanism extends the concept of monitors by allowing to wait for fulfillment of logical expressions composed of elementary events. The paper describes the implementation and illustrates it with examples of usage.

Key words:
Java, concurrency, synchronization, monitors

1. Introduction

One of the strengths of the Java language [1] is that it provides facilities for cross-platform concurrent programming. These facilities consist of means for creating multiple execution threads and for synchronization of their work. Java’s native synchronization mechanisms are however very low level. They are not only far away from the most elegant Dijkstra’s guarded commands proposal [3] but also from the simple Hoare’s monitors proposal [4]. Keedy [5] describes how the latter model fails when we want to delay a task until an alternative or conjunction of elementary events is fulfilled.

Gorazd and Kotulska [6] suggest a notation that allows to effectively suspend a task (inside a monitor-like construct) until a composite logical expression of the fulfillment of elementary events has been evaluated as true. The paper presents an implementation of the mentioned solution as a Java library called SynchExpressions 1.

2. Standard Java synchronization mechanisms and the SynchExpressions library

As mentioned in the introduction, Java’s native synchronization mechanisms are rather low level. A lock is associated with every Java object, and threads can acquire it by executing a synchronized block. Only one thread is allowed to own a particular lock at any given time. Once a thread gets object’s lock’s ownership, it can suspend its execution by calling the wait() method on the object, or notify() some other, waiting, thread that it can continue its execution. Thus, Java implements a simplified concept of monitors.

Let us note that:
- this solution lacks the ability to declare multiple condition variables, making it difficult to selectively wake threads awaiting specific conditions,
- Java does not guarantee queuing of waiting threads—a thread to be notified is selected in an unspecified manner.

The SynchExpressions library removes these limitations by providing its own version of monitors, that:
- allows to declare synchronization entities (called elementary events);
- allows to suspend the current thread until a logical expression combining fulfillment of elementary events has been evaluated as true;
- the threads waiting for a particular event are arranged in a queue, so the order of waking is the same as the order of suspending;
- fulfillment of the given event will be signaled explicitly by an operation named notify().

Thus, the library does not only implements the classic monitor model, but also extends it by providing the possibility of waiting for fulfillment of compound logical expressions composed of elementary events. By raising the abstraction level of synchronization constructs the library makes it easier to write correct concurrent programs.

---

1 The library can be obtained by writing to author: fryz@ii.uj.edu.pl
3. SynchExpressions as a tool for basic synchronization problems.

As the first step we suggest to analyze the solution of the classic bounded buffer problem. Here, multiple threads can insert elements into a buffer and extract elements from it. The elements have to be extracted in the same order as they were inserted. When the buffer is empty (no elements have been inserted or all have been extracted) and a thread requests extracting an element then the thread has to be suspended until new elements arrive. Similarly, an attempt to put a new element into a full buffer has to be blocked.

```java
public class Buffer extends Monitor {
    private byte[] buf;
    private int inIdx, outIdx;
    private int capacity, size;
    public Buffer(int capacity) {
        this.capacity = capacity;
        buf = new byte[capacity];
        size = 0;
        inIdx = 0;
        outIdx = 0;
    }
    protected void declareEvents() {
        declareEvent("NOTEMPTY");
        declareEvent("NOTFULL");
    }
    public synchronized byte get() {
        if( size == 0 ) {
            wait("NOTEMPTY");
        }
        byte res = buf[outIdx++];
        outIdx = (outIdx + 1) % capacity;
        size--;
        notify("NOTFULL");
        return res;
    }
    public synchronized void put(byte b) {
        if( size == capacity ) {
            wait("NOTFULL");
        }
        size++;
        buf[inIdx] = b;
        inIdx = (inIdx + 1) % capacity
        notify("NOTEMPTY");
    }
}
```

Some comments to the example:

- The Buffer class extends SynchExpressions library’s Monitor. The latter class is a cornerstone of the library. It provides support for declaring events, issuing them, and for waiting for fulfillment of expressions.
- The declareEvents() method introduces the elementary events definition for the created monitor;
- Both put() and get() methods are declared as synchronized. This is necessary due to the way SynchExpressions library’s mechanisms are implemented with Java’s synchronization primitives. Generally, all the methods of Monitor’s subclasses should be synchronized, except for the methods that are always called from within other methods—one such example is declareEvents(), which is only called from Monitor’s constructor and thus it is not required to be synchronized;
- The threads are suspended by a call to wait(...) and resumed by calling notify(...). These methods take as arguments, respectively, the expression to be waited for and the event to be issued. In the above example, the expressions are simply elementary events, but they can take a more complicated form.

4. Monitors and events

All the library’s facilities are provided through protected methods of the Monitor class. Thus, it is necessary to extend this class in order to use them. Monitor maintains a structure for declared events, called synchronizing expressions, history of the notified events and offers the methods to modify it.

4.1 Free and essential events

In the most of the synchronization problems (like the one mentioned above) we assume that event notifications cannot be lost (from now on we will call this kind of events essential events), otherwise the synchronization fails. The introduction of synchronization conditions using both and and or operators can violate this assumption. Let us consider the following statement:

```
wait("QA and QB or QC");
```

If both ‘QA’ and ‘QC’ events have been notified sequentially, then the thread executing the wait operation will be resumed, but the ‘QA’ event has been superfluously consumed. SynchExpressions offers here two possibilities:

- the expression can be dynamically reduced in the way that excludes losing the notifications,
- one can declare ‘QA’ and ‘QB’ as free events whose notification can be lost.

In the first case, after the essential event ‘QA’ is received, the expression will be reduced to just “QB”, so ‘QC’ event notification will be relayed to another thread. The complete semantics of this reduction is presented in [6] and in the documentation of the library. Let us however note that no reduction is necessary when only one type of
operators (either and or or) has been used in the wait expression.

In the real world, one can find examples of events that appear repetitively (e.g. clock signals) and can be lost without any negative consequences to synchronization; such events are the motivation for introduction of free events.

4.2 Declaring events

A concrete subclass of Monitor must inform the base class what events it is going to manage. This is accomplished by overriding an abstract method declareEvents. In its body (and nowhere else) the methods declareEvent and declareFreeEvent can be called in order to declare the appropriate type of event. These methods take the name of the event to be declared as their argument and (optionally) return an instance of the Event class, which can be used as a handle to the newly declared event.

```java
public synchronized void someMethod() {
    Event qa = getEventByName("QA");
    notify(qa);
    notify("QB");
    boolean deliveredC = notify(qc);
    ...
}
```

It is possible that no thread is currently interested in an issued event, so the thread notifying an event can check whether the event it generated has been consumed. For this reason notify() returns a boolean value—it's true if the event has been consumed; otherwise it's false.

5. Waiting for fulfillment of conditions

Suspending execution of threads until some desired conditions are met is the basic functionality of the SynchExpressions library. In order to suspend its execution, a thread has to call the wait() method of the appropriate Monitor. The expression for whose fulfillment we want to wait has to be provided as a String argument. For example:

```java
wait("QA and QB or QC rep 2");
```

causes suspending the execution until either events QA and QB are issued or event QC is issued twice.

5.1 A grammar for wait expressions

The syntax for correct wait expressions is given below:

```
expression ::= disjunction
disjunction ::= conjunction (or conjunction)*
conjunction ::= simpleExpr (and simpleExpr)*
simpleExpr ::= ID (rep POSITIVEINTEGER)?
```

Symbols ID and POSITIVEINTEGER denote, respectively, a name of one of the events declared for the Monitor and a positive integer defining number of repetitions of this event required for resuming the thread.

The POSITIVEINTEGER must be a constant—waiting for a number of event occurrences determined by a value of a variable is not directly supported. Such behavior can however be simulated by dynamic construction of the String representing the wait expression. For instance:
```java
int i=3;
wait("QA rep "+i);
```

### 5.2 Preparsed expressions

The process of parsing a String representing an expression is relatively expensive. Thus, if the program requires frequent waiting for the same expression, it is desirable to reduce this overhead.

For this reason, the library provides a facility to create expressions in preparsed form, which can be used for multiple wait() calls. In order to create such a form, the `createParsedExpression()` method has to be called. It returns an instance of the `ParsedExpression` class that can be used instead of a String as an argument to `wait()`. The following example illustrates this:

```java
ParsedExpression pexpr = createParsedExpression("QA and QB or QC");
wait(pexpr);
```

Of course, it makes no sense to create a parsed expression before every wait call—usually they should be created only once, for example in the constructor.

A String representation of the `ParsedExpression` can be retrieved using the `toString()` method. Additionally, it is possible to factor a `ParsedExpression` into objects representing its respective parts. This ability is provided by the `getAlternatives()` method. It returns an array of `Disjunct` objects, each representing one of the possibilities that can wake a thread waiting for fulfillment of the expression. Let’s look at an example:

```java
ParsedExpression pexpr = createParsedExpression("QA and QB or QC");
Disjunct[] alters = pexpr.getAlternatives();
```

Here, a two element array is created – its first element represents the QA and QB alternative and the second one the QC part.

### 5.3 Determining the cause of resumption

The possibility of waiting for fulfillment of an alternative of conditions makes it uncertain for the resumed thread why exactly it has been woken. In some circumstances, this knowledge can be vital to determine the further course of actions. Thus, it is necessary to make it possible for the thread to check which of the wait expression’s alternatives caused the thread to be awaken.

In order to provide this information, the `wait` method returns an instance of the `Disjunct` class as its result. It can be inspected through calling `toString()`, or it can be compared with elements of an array returned by the `getAlternatives()` method of `ParsedExpression`. This comparison is performed by the `equals()` method.

The following code demonstrates how these mechanisms can be used:

```java
ParsedExpression pexpr = createParsedExpression("QA or QB");
Disjunct[] alters = pexpr.getAlternatives();
Disjunct cause = wait(pexpr);
if(cause.equals(alters[0])) {
    // resumed by QA
} else if (cause.equals(alters[1])) {
    // resumed by QB
}
```

### 6. The final example

#### 6.1 Problem description

The following example outlines the properties and expressiveness of the library. The example concerns the situation where several threads produce some partial results (there are several types of these). Then, other threads consume the partial products to assemble a final one.

To be more specific, let us assume that there are 3 types of partial results—A, B and C. Also, there are 2 kinds of final products, the first one is a result of combining A with B, and the second one needs B and C.

Furthermore, we assume that the relative speed at which A, B and C are produced cannot be determined in advance, so it is desirable for the ‘assembler’ threads to be universal (capable of producing both kinds of final products, depending on the resources currently available.)

#### 6.2 SynchExpressions solution

```java
public class AssemblyMonitor extends Monitor {
    public void declareEvents() {
        declareEvent("A");
        declareEvent("B");
        declareEvent("C");
        declareFreeEvent("READY");
    }
    public static final int PRODUCT_1 = 1;
    public static final int PRODUCT_2 = 2;
    public synchronized void producedA() {
        boolean delivered=notify("A");
        while(!delivered) {
            wait("READY");
            delivered=notify("A");
        }
    }
    ... // the same for B and C ...
    public synchronized int waitForWork() {
        while(notify("READY")) {
            // wake producers
        }
    }
```
Disjunct res=wait("A and B or B and C");
if(res.toString().equals("A and B")) {
    return PRODUCT_1;
} else {
    return PRODUCT_2;
}
}

7. Conclusions

The low level nature of Java’s synchronization has been identified as a weakness and many solutions were proposed. Most of them revolve around the idea of extracting frequently used synchronization patterns, like blocking queues and worker thread pools, and providing their implementation through libraries. A notable example is Lea’s Concurrency Utilities package [2], which has been included as a part of standard library in release 1.5 of Java.

The SynchExpressions library attempts to solve this problem in a different way. Instead of solutions to specific synchronization scenarios it provides a more expressive general-purpose mechanisms.

References

Lukasz Fryz received the M.S. degree in Computer Science from Institute of Computer Science, Jagiellonian University in 2002. Since 2002 he is a Ph. D. student there. His research interests include graph grammars, distributed computing and visual languages.

Leszek Kotulski received:

- the M.S. degree in Computer Science from Institute of
  Computer Science, Jagiellonian University in 1979,
- the Ph.D. degree in Computer Science from AGH University of
  Science and Technology 1984,
- DSc degree in Theoretical Computer Science from Wroclaw
  University of Technology in 2002.

He works as Associate Professor in Computer Science Department, Jagiellonian University. His research interests include graph grammars, foundation of distributed computing, agents systems and software development methodology.