JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5. No. 7, September-October 2006

The JAC System: Minimizing the Differences between Concurrent and Sequential Java Code

Klaus-Peter Löhr, Freie Universität Berlin
Max Haustein, Freie Universität Berlin

Abstract

JAC is a Java extension for high-level concurrent programming, meant to hide the notions of threads and synchronization statements from the programmer. Putting into practice the concept of concurrency annotations suggested for Eiffel some time ago, one of JAC's main assets is its support for minimizing the differences between concurrent and sequential implementations of objects and computations. The paper focuses on this aspect of JAC, presenting examples of successful applications of the annotations and pointing out remaining weaknesses.

1 INTRODUCTION

Java was created as a concurrent language right from the beginning. Its concurrency features, however, did not reflect the state of the art in concurrent object-oriented programming at that time [Briot et al. 98]. A rather conservative approach was taken, featuring explicit threading and a version of monitors that was even less safe than previous monitor concepts. Concerning the interplay between concurrency and inheritance, Java is a prime example of a language for which the infamous inheritance anomaly is the rule rather than the exception. And regarding C#, Microsoft's answer to Java, one is led to wonder why the designers of the language did not take a different road.

Java tries to alleviate the problems of concurrent programming by library support. For instance, a properly synchronized version of an (unsynchronized) `java.util. TreeSet` object is created by `Collections.synchronized-SortedSet(new TreeSet(...))`. The `java.util.concurrent` classes adopted for Java 1.5 support typical patterns and solutions for common concurrent programming problems [Lea 04]. Several Java extensions meant to support higher-level concurrency models have been suggested (see, e.g., [Bacon et al. 00] [Felber/Reiter 02] [Itzstein/Kearney 02] [Milicia/Sassone 02] [Olsson/Keen 04] and others). In addition, the aspect-oriented community, viewing synchronization just as an aspect to be added to business logic, has suggested weaving synchronization code into a given piece of sequential code [Kiczales et al. 01]. A similar approach is encapsulating objects in what has been called composition filters [Aksit et al. 94], synchronization rings [Holmes 99] or qualifying/ qualified types [Keedy et al. 02] [Keedy et al. 05].

The high-level concurrency extensions to Java we are presenting here are focussed on 1) declaratively stating concurrency aspects, thus minimizing the differences between concurrent and sequential code, and 2) minimizing inheritance anomalies. The extensions come as *concurrency annotations*, emphasizing the fact that they can be either obeyed – giving rise to concurrent semantics – or ignored – giving rise to sequential semantics. Objects of the same class but with differing semantics can even coexist in one program. The extended language is called JAC – for Java with Annotated Concurrency – and draws from previous experience with a design for concurrency annotations for Eiffel [Löhr 93]. JAC has been implemented through a precompiler that generates regular Java code. The system and its documentation are freely available from http://www.inf.fu-berlin.de/inst/ag-ss/jac. The reader is referred to the documentation and to [Haustein/Löhr 06] for checking on semantic details that are omitted from this paper due to space limitations.

We will begin our presentation by giving a first flavour of JAC in section 2: we investigate cases where some given sequential code has a natural concurrent semantics as well; we also point out the limits of this view. Section 3 explores JAC from the opposite direction: we argue that library classes should come as concurrent (JAC) code, but in such a way that they can be used as sequential code as well, without any performance penalty. Section 4 presents a case study and argues that there is a close connection between the concurrent semantics and the sequential semantics of JAC code. We conclude with sketching our implementation (section 5) and trying an assessment of the JAC approach (section 6).

2 CONCURRENT SEMANTICS OF SEQUENTIAL PROGRAM TEXT

In a concurrent system, objects that are shared among concurrent activities have to be synchronized in a proper fashion. Assuming for now that concurrency is present in a program (how this is achieved in JAC will be explained later), we look at synchronization. Synchronization measures can be roughly classified as either exclusion synchronization or condition synchronization. We begin with exclusion synchronization.

Exclusion synchronization

Consider a typical Java collection class, say, the aforementioned `java.util.TreeSet`. To avoid performance penalties for sequential usage, this class is not synchronized and thus cannot be used safely in a concurrent context (i.e., by more than one thread at a time). But instead of resorting to wrappers we can take a higher-level view and argue that the compiler should be able to add the required synchronization code, essentially establishing the object as a monitor. This is easier said than done, though: how can the compiler know that the object will be used by concurrent threads? It turns out that this is not possible without considerable modifications to the language; *Guava* [Bacon et al. 00] is a prime example of this approach.

A lightweight, if not fully automatic, alternative is to rely on the programmer's knowledge about the object's intended usage. This is what JAC does: if a statement containing an object creation expression is marked with a *control annotation*, access to that object will be synchronized, as explained below. For instance, given a class `SimpleSet implements java.util.Set`, such a statement could read

```
controlled Set jobs = new SimpleSet();
```

Note that this annotation is *not* applicable to collection classes of present Java systems, because JAC relies on *source* code transformation. Also note that the annotation *may or may not* be obeyed, depending on the context, as discussed in section 4.

`controlled` is slightly reminiscent of the `separate` keyword that has been suggested for Eiffel/SCOOP [Meyer 97] [Nienaltowski et al. 03] [Fuks et al. 04]. There are major differences, though, to be discussed below.

Condition synchronization

Buffers have always been a favorite study object in concurrent object-oriented language design. Sequentially used *queues* and concurrently used *buffers* have almost identical functionality. They behave differently only when a client tries to remove an entry from an *empty* queue/buffer (given infinite capacity). Here again we can argue that the compiler should be responsible for providing code that either raises an exception or just causes a delay, depending on whether the object is used in a sequential or in a concurrent setting.

In a first approximation, this suggests that methods in classes envisaged for concurrent usage should not raise exceptions explicitly but should rather be equipped with a precondition, as known from Eiffel's `requires` clause. Establishing the appropriate object behaviour – according to the control annotation – would again be left to the compiler. The precondition would be interpreted as a *guard* for possible delays in the concurrent case. A tentative syntax is shown in this version of a queue/buffer class:

```
class Queue<Item> { // infinite queue/buffer
public void append(Item i) {.....}
public Item remove() pre length>0 {.....}
.....
}
```

The rationale is, of course, that delaying makes sense because the guard refers to the object's state and will become true (hopefully ;-) in due time. By implication, guards referring to a method's arguments only should *always* raise exceptions if not valid.

Taking a precondition's referral to the object's state as a necessary and sufficient condition for delaying is a fallacy, though. There are cases where an exception, not a delay, is called for even in a concurrent setting. Consider the example of a "cloak room" where objects can be deposited in exchange for a "ticket"; when picking up the object later, the ticket has to be presented:

```
class CloakRoom {
public Ticket deposit(Object o) pre !full {
    return new Ticket(o); }
public Object pickup(Ticket t) pre isValid(t) {
    return t.getObject(); }
..... // storage, nested class Ticket, etc.
}
```

Tickets are not reused; a ticket is invalidated by the very act of picking up the corresponding object. The preconditions both of `deposit` and of `pickup` are dependent on the state of their `CloakRoom` object. But if the check for `isValid(t)` fails, an exception has to be raised in any case, irrespective of whether the cloak room is used sequentially or concurrently. We conclude that the idea to just furnish Java with a precondition clause the semantics of which will depend on the control annotation is too simple-minded. So we reject the tentative `pre` and look for a better solution.

3 SEQUENTIAL SEMANTICS OF CONCURRENT PROGRAM TEXT

The delay annotation

Taking into account the arguments given above, JAC distinguishes between preconditions and guards, featuring two additional annotations, a *precondition annotation* and a *guard annotation*. The corresponding keywords are `if` and `when`, respectively, and the cloak room example becomes

```
class CloakRoom {
public Ticket deposit(Object o) when !full {
    return new Ticket(o); }
public Object pickup(Ticket t) if isValid(t) {
    return t.getObject(); }
..... // storage, nested class Ticket, etc.
}
```

If an object of this class is *controlled*, `deposit` may cause a delay (the guard being re-evaluated upon each method return) and `pickup` may raise an exception (`ViolatedPreconditionException`). If the object is not controlled, the semantics of `if` and `when` are identical. Note that it is not unusual for a method to have both annotations, as in

```
public Ticket deposit(Object o) if o!=null when !full {
    return new Ticket(o); }
```

Having introduced these annotations we can no longer claim that any given sequential class code has a natural concurrent semantics. *But the inverse argument is still valid.*

Exclusion synchronization revisited

JAC supports a kind of weak exclusion mechanism that is a declarative generalization of read/write locking. A method m can be declared “compatible with methods x,y,…” by means of a *compatibility annotation*, as in

```
class MyMap<Key,Value> implements Map<Key,Value> {
public void add(Key k, Value v) {.....}
public Value lookup(Key k) compatible lookup(Key) {...}
.....
}
```

The keyword `compatible` is followed by a comma-separated list of method signatures for uniquely identifying the methods in question. “x is compatible with y” means that there are no race conditions between x and y, so their executions can safely overlap in arbitrary ways. The simple example given above solves the reader/writer problem between `lookup` and `add`: the method `add` is not compatible with any other method; `lookup`s can be executed concurrently.

Compatibility is a non-reflexive, non-transitive, but *symmetric relation*. Due to the symmetry, redundant compatibility annotations may be omitted. This blends well with inheritance and avoids many pitfalls with respect to inheritance anomalies. For instance, extending the class `MyMap` towards

```
class ExtendedMap<K,V> extends MyMap<K,V> {
public boolean contains(V v) compatible contains(V),
                         lookup(K) {...}
}
```

does not require any overriding of inherited methods. Also note that compatibility is orthogonal to the notions of precondition and guard, and a method may well be furnished with all these annotations. Constructing a version of the buffer example where `append` is compatible with `remove` is left as an exercise.

The compatibility annotation - as opposed to Java’s `synchronized` modifier - is attractive because omitting it means “erring” on the safe side: the result is just stricter exclusion. In many cases, the programmer who designs a class may defer adding compatibility annotations - or just not care about them - without introducing race conditions.

The compatibility annotation, just like the other annotations, is ignored for objects that are not controlled – because synchronization is not required in this case. This means that the introduction of the compatibility annotation does not invalidate our claim that concurrent JAC code has a natural sequential semantics.

Generating concurrency

Concurrency originates from *active objects* in JAC. A method can be either of

- synchronous,
- asynchronous,
- autonomous.

A normal Java method is *synchronous*. A method annotated with `async`, like in

```
public async Object method(...) {.....} ,
```

is executed in an *asynchronous* fashion, i.e., concurrently with the caller which continues right after the invocation; if the method is not `void`, a *future* object is returned. An *autonomous* method, annotated `auto` instead of `async`, like in

```
protected auto void method() {.....} ,
```

is executed without ever being invoked, and is repeated indefinitely (possibly subject to delays if guarded). A class can feature an arbitrary mixture of synchronous, asynchronous and autonomous methods. If it has at least one asynchronous or autonomous method, it is called an *active class*. A JAC active object is a controlled object of an active class. Note that the designer of the class has to be careful when declaring a method asynchronous, ensuring that interleaving activities of caller and callee, whatever the caller may do (!), must not have undesirable effects. We will come back to this issue below.

Benefits of just having special methods, instead of introducing a notion like thread or task, include - again - minimal deviation from sequential code and, by implication, minimal susceptibility to inheritance anomalies. `async` and `auto` are again ignored for non-controlled objects, leaving behind sequential Java code with unaltered business logic. So here we have another example of concurrent code that is reusable in a sequential setting, fitting this section's theme of "sequential semantics of concurrent program text".

However, comparing the *context* of an active object (i.e., controlled) to that of a corresponding passive object (i.e., non-controlled), we will necessarily see more differences than for objects of non-active classes. A non-controlled object will of course never execute its autonomous methods. Explicit invocation of such a method (visibility allowing) is fine, but a typical concurrent context for the active version will of course *not* invoke that method (while not forbidden, is has no effect).

The situation is less problematic with asynchronous methods. It is sometimes argued, independently of a specific programming language, that executing a given procedure/method in an asynchronous rather than a synchronous fashion is a natural way of parallelizing sequential code. This is a fallacy, though, because the semantics of synchronous and asynchronous execution *are different*. With synchrony, a caller continuing after the call can rely on the effects brought about by the method; with asynchrony it cannot, and the danger of races looms. So we avoid the claim that a synchronous method has the asynchronous interpretation as its natural concurrent semantics. But we do state that it is safe to argue the other way around: given an asynchronous method, syntax and semantics remain the same for its synchronous invocation in a sequential setting (unless, of course, there is explicit cooperation among caller and callee via shared objects featuring condition synchronization).

4 SIMPLE CASE STUDY: A PARALLEL ASSEMBLER

We present a simple but non-trivial example of a concurrent JAC program that can be transformed into a sequential version with ease. While this may not be too exciting in itself, the purpose rather is to see the annotations in action and to appreciate the proximity of sequential versus concurrent readings of the classes involved. Ultimately, it is to convince the reader that it makes sense to design concurrent code that can be used (or reused) both under a concurrent and under a sequential interpretation.

The two-pass assembler presented below includes a *parser* and a *code generator*. As forward references are supported, the final machine code equivalent of an assembler statement containing a symbolic address can only be generated after the symbol has been found by the parser. A sequential assembler would first do a complete parse and then generate the code (or do a one-pass translation right away). But a certain degree of concurrency between parser and generator is possible, with two constraints: 1) the generator must not get ahead of the parser when processing the statement sequence and 2) the generator may be slowed down further by forward references not yet resolved by the parser. Parallel execution on a multiprocessor would promise a certain speedup, depending on the presence of forward references (and of course on the synchronization overhead).

Our parallel assembler gets its input from a source file f (a command line parameter) and produces machine code in a file f.out. The parser and the generator share the intermediate code – an object of class `Code` – and the symbol table – an object of class `MapImpl` – produced by the parser. The translation process is set up by the `main` method of class `Assembler`:

```
import java.io.*;

public class Assembler {
    static String input;
    static FileOutputStream output;
    static Code intermediate;
    static MapImpl symbolTable;

public static void main(String[] arg) throws IOException {
    input = arg[0];
    output = new FileOutputStream(input + ".out");
    controlled intermediate = new Code(new FileReader(
                        input));
    controlled symbolTable  = new MapImpl();
    controlled Assembler asm = new Assembler();

    asm.parser();
    asm.generator();
    System.exit(0);
    }
.....
```

For establishing concurrent operation of parser and generator, the `parser` method is declared asynchronous. So the class introduced above continues like this:

```
async void parser() {
   for(int ip=0;;) {
     Statement s = null;
     try { s = intermediate.addStatement(); }
     catch(SyntaxErrorException e) {...}
     catch(IOException e)     {...}
     if(s==null) break;
     ...
     if(s.label != null) ...
       symbolTable.put(s.label, new Integer(ip));
     ip += s.length;
     }
   symbolTable.close();
}
```

All error handling (for missing labels, name clashes etc.) has been omitted here for ease of presentation. `addStatement` reads and parses a source statement from the input file and appends its objectified version to the intermediate code. If the statement is labelled, `put` adds the appropriate entry (with instruction pointer `ip`) to the symbol table. The parser indicates the end of information flow into the symbol table by issuing a `close`; we will come back to this below.

For ensuring that parser and generator can indeed proceed concurrently, the `generator` method is declared compatible with the `parser` method. (Other designs would be possible, e.g., introducing separate parser and generator objects.). So the `Assembler` class concludes as follows:

```
void generator() compatible parser() {
    for(;;) {
      Statement s = intermediate.nextStatement();
      if(s==null) break;
      ...
      Integer addr = (Integer)symbolTable.get(s.target);
      int address;
      if(addr != null) address = addr.intValue();
      ...
      byte[] instruction = s.generate(address);
      try { output.write(instruction); }
      catch(IOException e) {...}
      }
    System.out.println("Done.");
    }
}
```

Getting the next intermediate code statement using `nextStatement` may block if that statement is not available yet. Similarly, a symbol table lookup using `get` may block if the symbol in question is not present yet. Note that a missing label would cause a hangup in the generator if the parser would not signal that parsing is done and no more symbol table entries are to be expected. This is why the parser uses `close`, as shown above. The `get` method (see below) has to recognize this.

The class for managing the intermediate code is simple enough:

```
import java.util.*;
import java.io.*;
class Code { ...
    Vector code = new Vector();
    int current = 0; // instead of iterator
public Statement addStatement() throws SyntaxErrorException,
                    IOException {
    Statement statement;
    ... // read and parse line, transform into statement,
      // add statement to vector of statement objects,
      // ... null if EOF !
    return statement;
    }
public Statement nextStatement() when current<code.size()
                 compatible addStatement() {
    return (Statement)code.get(current++);
    }
}
```

The careful reader will notice that this class does not require any exclusion measures at all *when used in the special context presented here*. We could achieve the desired effect by just declaring `concurrent class Code { ..... }`; this is equivalent to stating that all methods are mutually compatible, so we would not have to struggle with adding compatibility annotations.

The last class to be considered is `MapImpl`, used for the symbol table object:

```
import java.util.*;
class MapImpl {
    Map map = new java.util.HashMap();
    boolean closed = false;
    Object lastKey, lastValue;
public Object get(Object key) when containsKey(key)||closed
{
    boolean ok = lastValue != null &&
          key.equals(lastKey)?true:containsKey(key);
    return ok ? lastValue : null;
    }
public Object put(Object key, Object value) {
    Object res = map.put(key,value);
    return res;
    }
public boolean containsKey(Object key) {
    lastKey  = key;
    lastValue = map.get(key);
    return lastValue != null;
    }
public void close() { closed = true; }
}
```

The `MapImpl` code, although largely relying on `HashMap`, is not quite trivial. This is due to the fact that evaluating the crucial part of the `get` method's guard, `containsKey`, virtually amounts to the lookup effort itself. To avoid duplicating this effort, `containsKey` does a complete lookup and remembers the result. Note that ignoring the well-known caveat against side-effects of preconditions or guards, as seen here, must go together with a thorough check that no harm is done. – We omit the class `Statement`. Its code is simple, and no new insight is gained from it.

Having presented the parallel assembler, we would like to emphasize two points. First, deleting all JAC annotations in the given code will not only produce a syntactically correct (sequential) Java program, but this program is also free of runtime errors *and produces the same result*. We do not claim that this is the case for any imaginable JAC program (actually, it is not). But it should be obvious that the semantics of the sequential version of a JAC program is so close to its concurrent semantics that this version can either be used right away or after minimal modifications.

The second point may be more important: consider library classes, not complete programs. A library class of a JAC system should always be furnished with the appropriate annotations for a concurrent environment (variants for restricted concurrency may also be helpful, e.g., a buffer for only one producer and one consumer). If such a class is used for creating a non-controlled object, no synchronization overhead will be incurred; the object will just ignore all the annotations. The emphasis is on *ignore* rather than *delete* (as above). Thanks to the control annotation, objects of both kinds can even coexist in one program.

A last remark is in order. The control annotation, like all annotations, is itself subject to control: it is ignored within non-controlled objects. Static methods are not subject to concurrency control; they serve as “anchor points” from which all concurrency-related behaviour emanates. This has interesting ramifications which are beyond the scope of this paper, though.

5 IMPLEMENTATION ISSUES

For the current version of the precompiler, the annotations do not come as keywords but as special *Javadoc tags* (which are ignored by the Java compiler). Aesthetically, this is less pleasing; it has two advantages, though: 1) any JAC program is a valid sequential Java program, so the assembler mentioned above works either in parallel (JAC+Java compilation) or sequentially (Java compilation); 2) ease of implementation (see below). Here is the actual version of the aforementioned `nextStatement` method

```
/**
 * @when current<code.size()
 * @compatible addStatement()
 */
public Statement nextStatement() {
    return (Statement)code.get(current++);
    }
```

The precompiler uses *Barat*, a compiler front-end for Java [Bokowski 98]. Barat parses Java source code files and builds an *abstract semantics graph* (ASG) containing name and type analysis information. The ASG is traversed using the well-known visitor pattern. Barat provides an OutputVisitor that rewrites the source code parsed from a file using a default syntax. Formatting issues are not covered by the ASG. The precompiler extends the standard visitor to insert the modifications implementing the semantics expressed by the annotations. Additional classes are generated on demand. Note that Barat and, by implication, JAC do not support Java 1.5 (as of August 2005). So several of the code examples given above, exhibiting generics, are dreams of the future rather than reality.

Any class processed by the precompiler is transformed into another class that can be used both for controlled and for non-controlled objects. Modified constructors, parameterized according to the presence or absence of a control annotation, determine the behaviour of the object.

Several shell scripts supporting easy usage of the system are available. In addition, the *ant* tool can be used in connection with an easily adaptable build.xml.

All of this can be found on JAC's website, http://www.inf.fu-berlin.de/inst/ag-ss/jac, together with documentation for the language. We repeat our earlier remark that space does not allow for a complete language description here.

6 ASSESSMENT AND CONCLUSION

Separating business logic from concurrency issues is an important principle for high-level concurrent programming. This is not specific to Java dialects, or even to object orientation in general; but it has the potential of blending especially well with inheritance, mitigating inheritance anomalies.

Several examples of applying that principle to Java have been mentioned in the introduction. JAC is unique among them in its extremely lightweight mechanism for choosing between the sequential and the concurrent version of a class – the annotation `controlled`. Its closest relative is *Eiffel/SCOOP's* `separate` keyword, recently also adopted for Java [Morales 05]. But note that `separate` (by design) implies 1) asynchronous operations, 2) mutual exclusion of all operations, and 3) indiscriminately interpreting all state-related preconditions as guards (which we identified as unsound in section 3.1).

As opposed to this, JAC uses additional annotations, in order to give the programmer more degrees of freedom. This greater flexibility comes at a price: the annotations are not foolproof. While we may omit most annotations without causing much harm, this is *not* true for the control annotation. For instance, forgetting to declare `controlled` an object that is shared among several concurrent activities would wreak havoc in most cases. Eiffel is safer in this respect. Among the Java dialects, *Guava* enjoys the same degree of safety [Bacon et al. 00] (but does not enjoy the flexibility of deciding about an object's semantics at declaration/creation time).

The existing Java dialects suffer from inheritance anomalies to different degrees. It should always be remembered that describing an object's synchronization properties in a centralized fashion, detached from the code proper, say, with a path expression or the like, is asking for problems [Briot et al. 98]. While looking attractive at first sight, such an approach is easily prone to inheritance anomalies because extending a class will necessarily require revisiting, and probably modifying, the synchronization expression. JAC's rules for inheritance (described in detail in the documentation) eliminate all inheritance anomalies except the history-related anomaly.

Last but not least, we view `async` and `auto` as valuable assets of JAC. They hide the notion of threads, blend extremely well with inheritance, and offer the programmer a choice: while `async` and `auto` *can* be used to simulate each other, the experience is rather unpleasant, and there is enough of a pragmatic difference to warrant supporting both.

REFERENCES

[Aksit et al. 94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa: "Abstracting object interactions using Composition Filters", Proc. *ECOOP '93 Workshop on Object-Based Distributed Programming*, LNCS 791, pp. 152-184, Springer, 1994.

[Bacon et al. 00] D.F. Bacon, R.E. Strom, A. Tarafdar: "Guava: a dialect of Java without data races", Proc. *Conf. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA '00), pp. 382-400, October 2000.

[Bokowski 98] B. Bokowski, A. Spiegel: "Barat – a front-end for Java", TR B-98-09, Fachbereich Mathematik und Informatik, Freie Universität Berlin, December 1998. See also http://barat.sourceforge.net

[Briot et al. 98] J.-P. Briot, R. Guerraoui, K.-P. Löhr: "Concurrency and distribution in object-oriented programming", *ACM Computing Surveys*, vol. 30, no. 3, pp. 291-329, September 1998.

[Felber/Reiter 02] P. Felber, M.K. Reiter: "Advanced concurrency control in Java", *Concurrency and Computation: Practice and Experience*, vol. 14, no. 4, pp. 261-285, April 2002.

[Fuks et al. 04] O. Fuks, J. Ostroff, R. Paige: "SECG: The SCOOP-to-Eiffel code generator", *Journal of Object Technology*, vol. 3, no. 10, pp. 143-160, November - December 2004, http://www.jot.fm/issues/issue_2004_11/article3

[Haustein/Löhr 06] M. Haustein, K.-P. Löhr: "JAC – Declarative Java concurrency", Concurrency and Computation: Practice and Experience, vol. 18, no. 5, pp.519-546, 2006. http://www3.interscience.wiley.com/cgi-bin/abstract/112126170

[Holmes 99] D. Holmes: "Synchronization Rings – Composable Synchronization for Object-Oriented Systems", PhD thesis, Mcquarie University, Sydney, 1999.

[Itzstein/Kearney 02] G.S. Itzstein, D. Kearney: "Applications of Join Java", Proc. *7. Asia-Pacific Computer Systems Architecture Conf.* (ACSAC 2002), Melbourne, 2002.

[Keedy et al. 02] J.L. Keedy, G. Menger, C. Heinlein, and F. Henskens: "Qualifying Types illustrated by synchronisation examples", Proc. *Objects, Components, Architectures, Services and Applications for a Networked World, Int. Conf. NetObjectDays* (NODe 2002), LNCS 2591, pp. 330-344, Springer, 2003.

[Keedy et al. 05] J.L. Keedy, K. Espenlaub, Ch. Heinlein, G. Menger, M. Evered: "Statically qualified types in Timor", *Journal of Object Technology*, vol. 4, no. 7, pp. 115-137, September/October 2005. http://www.jot.fm/issues/issue_2005_09/article5

[Kiczales et al. 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An overview of AspectJ", Proc. *15. European Conf. on Object-Oriented Programming* (ECOOP '01), LNCS 2072 , pp. 327-353, Springer, 2001.

[Lea 04] D. Lea: "The java.util.concurrent synchronizer framework". Proc. *Workshop on Concurrency and Synchronization in Java Programs* (CSJP '04), pp. 1-9, St. Johns, July 2004. http://www.podc.org/podc2004/csjp-proceed.pdf

[Löhr 93] K.-P. Löhr: "Concurrency annotations for reusable software", *Comm. ACM*, vol. 36, no. 9, pp. 81-89, September 1993.

[Meyer 97] B. Meyer: *Object-Oriented Software Construction*. Pearson, 1997.

[Milicia/Sassone 02] G. Milicia, V. Sassone: "Jeeg: a programming language for concurrent objects synchronization", Proc. *Joint ACM-ISCOPE Conf. on Java Grande* (JGI-02), ACM Press, 2002.

[Morales 05] F. Morales: "Eiffel-like separate classes", http://jdj.sys-con.com/read/36146.htm

[Nienaltowski et al. 03] P. Nienaltowski, V. Arslan, B. Meyer: "Concurrent object-oriented programming on .NET", *IEE Proceedings – Software*, vol. 150, no. 5, pp. 308-314, October 2003. See also http://se.ethz.ch/research/scoop.html

[Olsson/Keen 04] R.A. Olsson, A.W. Keen: "The JR Programming Language", Kluwer, 2004.

About the authors

Klaus-Peter Löhr is a professor of computer science and head of the Systems Software Laboratory at Freie Universität Berlin. His research interests include concurrent systems, distributed software architecture, systems security and software visualization. He can be reached at lohr@inf.fu-berlin.de

Max Haustein received his Dipl.-Inform. degree at Freie Universität Berlin. He is a research and teaching assistant in the Systems Software Laboratory, working on architecture and development of component-based software. He can be reached at haustein@inf.fu-berlin.de