

A Critique of Java for Concurrent Programming

Vijay K. Garg, *University of Texas at Austin*
Neeraj Mittal, *University of Texas at Dallas*

It's the year 2015. The number of processors in a commercial microprocessor chip has been increasing by a factor of two every year, so your desktop has at least a thousand processors. Furthermore, your desktop is connected to millions of other processors whose power you can tap into. Clearly, it won't behoove you, a computer scientist, to use a single processor to accomplish any programming task. You start writing your program in your concurrent programming language of choice—*Avaj*. Your mom, who used to be a Java programmer, happens to look over your shoulder and asks you how *Avaj* differs from Java. What will your reply be?

Java vs. Avaj

Here are some of our predictions.

`await(B)`

Java uses `wait()`, `notify()`, and `notifyAll()` constructs to support monitors. *Avaj* simply supports the construct `await(B)`, where B is any Boolean expression. When a thread T calls `await(B)`, T goes to sleep with the guarantee that if no other thread is in the monitor and B is true, then it will be awakened. Other threads don't

explicitly notify T when B could have become true. Instead, whenever any thread that could have made B true leaves the monitor, B is evaluated, and T is awakened if B evaluates to true.

Explicit notification in terms of `notify()` and `notifyALL()` constructs violates information hiding. A programmer, when writing a monitor's methods, must know all conditions that could become true when a thread executes a method of the monitor. This violates separation of concerns. Missing notification also results in deadlocks for many programs.

Avaj uses static analysis to determine when a thread should evaluate B . Moreover, Avaj allows the `await(B)` construct even in the context of message-passing programs where B might refer to the state of objects on remote nodes. In this case, for efficiency, Avaj restricts B to be in the disjunctive normal form of local predicates. Global-predicate detection algorithms¹ can be employed to detect B and wake the thread.

abort()

The *abort* concept has been around in database transactions for a long time. Supporting fault tolerance in databases is difficult without the ability to abort a transaction. This style of thinking still hasn't permeated general-purpose programming languages. In Java, you can handle exceptions, but the state of objects that have experienced a fault might be corrupted when the program receives an exception. What you frequently need is the ability to return to a state in which all objects are in good state and then either try the same method or an alternative.

The syntax for the Avaj `abort()` construct is simple. The programmer can call `abort()` inside any method. Such a method is *abortable*. The compiler ensures that enough state is saved so that the program can revert to the state in which the abortable method was called.

In message-passing programs, an abortable method might have sent some messages that would have to be undone. So, special care needs to be taken for handling messages sent by an abortable method directly or indirectly. The issues in implementing the `abort()` construct are quite similar to those investigated in the context of recovery schemes with optimistic message logging.¹

selective communication

The `selectivecommunication` construct in Tony Hoare's CSP (Communicating Sequential Processes) lets a process choose an appropriate interaction on the basis of Boolean guards.² It generalizes Edsger Dijkstra's alternative and repetitive command.³ This construct encourages programmers to write nondeterministic programs, which are inherently more fault tolerant than deterministic programs. If a choice leads to an unanticipated failure, then the system can try an alternative after aborting all the work done on the basis of the first choice. Writing nondeterministic programs in Java is cumbersome. Avaj's designers have recognized and remedied this shortcoming by providing the `selective communication` construct in the language.

invariant(B)

This Avaj construct allows a Boolean expression to be invariant in a program. If on execution of any method, B becomes false, then the system won't execute that method. If feasible alternatives exist, the system tries them. Otherwise, the program simply notifies the user and waits for instructions. This construct lets the programmer model integrity constraints such as "the account balance in a bank should never be negative" or "the dosage of the medicine should never exceed 1 g." Java supports assert statements. Invariant statements are like global asserts; they're always true before and after execution of any method of a given object.

As we mentioned earlier, in a message-passing program, B might refer to the state of remote objects. Checking the state of remote objects after every method invocation might not be feasible. For such programs, Avaj would employ *controlled reexecution*.¹ This technique executes the program in lookahead mode for some duration. If no violation of B appears, then that execution is committed; otherwise, the system analyzes whether additional synchronization can maintain B . If not, the system must try different choices for nondeterministic commands (or a different processing order of messages).

Conclusion

Such constructs would remove some programming burden at the expense of runtime overhead. We believe that improvements in programmers' productivity and programs' fault tolerance will outweigh the small overhead cost.

Acknowledgments

Vijay Garg's research was supported in part by NSF grants CNS-0509024, ECS-9907213, and CCR-9988225, and an Engineering Foundation fellowship.

References

1. V.K. Garg, *Elements of Distributed Computing*, John Wiley & Sons, 2002.
2. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985; www.usingscp.com.
3. E.W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.



Vijay K. Garg is a professor in the Electrical and Computer Engineering Department at the University of Texas at Austin and the director of the university's Parallel and Distributed Systems Laboratory. Contact him at garg@ece.utexas.edu.



Neeraj Mittal is an assistant professor in the Department of Computer Science at the University of Texas at Dallas. Contact him at neerajm@utdallas.edu.

Cite this article: Vijay K. Garg and Neeraj Mittal, "A Critique of Java for Concurrent Programming," *IEEE Distributed Systems Online*, vol. 6, no. 9, 2005.