Teaching Concurrency-Oriented Programming with Erlang

Ariel Ortiz
Information Technology Department
Tecnológico de Monterrey, Campus Estado de México
Atizapán de Zaragoza, Estado de México, Mexico. 52926
ariel.ortiz@itesm.mx

ABSTRACT

Teaching how to write correct programs is hard; teaching how to write correct concurrent programs is even harder. There is a desperate need for a better concurrency programming model than what most people are currently using. The Erlang programming language might be a step in that direction. This paper provides an overview of Erlang and how it has been successfully used to teach concurrency-oriented programming (COP) in a sophomore level course at the Tecnológico de Monterrey, Campus Estado de México.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming.*

General Terms

Design, Languages, Performance.

Keywords

Concurrency, Programming, Languages, Erlang, Parallel, Distributed, Functional, Multi-core.

1. INTRODUCTION

Concurrency is a difficult topic. So, as computer science educators, why do we bother with it? The CS2008 Review Taskforce [4] has an answer to this question:

> The development of multi-core processors has been a significant recent architectural development. To exploit this fully, software needs to exhibit concurrent behavior; this places greater emphasis on the principles, techniques and technologies of concurrency. ... Such a view implies that the increased emphasis on concurrency will not be a passing fashion but rather it represents a fundamental shift towards greater attention to concurrency matters. ... The increased ubiquitous nature of computing and computers represents a further factor in heightening the relevance of this topic and again there is every indication that this will only gain further momentum in the future.

Applications will increasingly need to be concurrent if they want to fully exploit continuing exponential CPU throughput gains. Herb Sutter, in his famous article entitled "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software" [16], states that concurrency is the next major revolution in how we write software:

> Different experts still have different opinions on whether it will be bigger than OO, but that kind of conversation is best left to pundits. For technologists, the interesting thing is that concurrency is of the same order as OO both in the (expected) scale of the revolution and in the complexity and learning curve of the technology.

Later in that same article, Sutter asserts the following:

> The mainstream state of the art revolves around lock-based[1] programming, which is subtle and hazardous. We desperately need a higher-level programming model for concurrency than languages offer today.

Joe Armstrong, the main designer of the Erlang programming language, agrees with this assertion, and proposes his solution [2]:

> Using essentially sequential programming languages to write concurrent programs is difficult and leads to the notion that concurrent programming is difficult; indeed writing concurrent programs in languages like Java or C++ which were designed for sequential programming is very difficult. This is not due to the nature of concurrency itself, but rather to the way in which concurrency is implemented in these languages.
>
> There is a generation of languages, which includes Erlang and Oz, in which writing concurrent programs is the natural mode of expression.

2. CONCURRENCY OVERVIEW

2.1 Defining Concurrency

A program is considered to be *concurrent* if it has more than one active execution context (a.k.a. thread of control). According to [15], concurrency has at least three important motivations:

[1]The term *lock-based* refers to synchronization mechanisms such as semaphores, monitors or mutexes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.
Copyright 2011 ACM 978-1-4503-0500-6/11/03 ...$10.00.

1. **To capture the logical structure of a problem.** Some programs are said to be *embarrassingly parallel*. This means that little or no effort is required to separate the problem into a number of independent tasks. Typical examples include: serving static files on a web server, brute force searches in cryptography, and rendering of computer graphics.

2. **To exploit extra processors, for speed.** In the past, multiprocessor[2] systems used to be expensive and rare. Today they are cheap and fairly common. Many laptops and even some netbook computers currently come with a dual-core CPU. To use these processors effectively, programs must generally be written with concurrency in mind [7].

3. **To cope with separate physical devices.** A program that runs on a group of computers scattered across a network is inherently concurrent. In a similar vein, embedded applications often have separate processors for each of several physical devices. Some examples: medical equipment, home automation, and automobiles.

The term *concurrent* describes a system in which at least two tasks may be underway at the same time. On a single core processor, concurrency can be achieved using time-shared preemptive threads or processes. A concurrent system is *parallel* if more than one task can be physically active at once. To achieve this, a multiprocessor system is required. A parallel system is *distributed* if its processors are associated with devices that are physically separated from one another. Under these definitions, "concurrent" applies to all three motivations. "Parallel" applies to the second and third, while "distributed" applies to only the third [15]. Erlang simplifies the construction of all these types of applications.

2.2 Concurrency-Oriented Programming

Concurrency-oriented programming (COP) is a term coined by Joe Armstrong [2]. He argues that a concurrency-oriented language should offer a good support for concurrency in the same way that an object-oriented language offers good support for objects. In COP, the programmer breaks up the solution of a problem into a number of parallel processes; a concurrency-oriented language should make this easy, while its runtime environment should make it inexpensive.

2.3 To Mutate, Or To Not Mutate

At the end of Part I of [7], there is a "concurrency cheat sheet" that summarizes a set of eleven rules that should be followed in order to write good concurrent code. Although these rules were written specifically for Java, they actually apply to other object-oriented languages that rely on threads and shared memory:

1. *It's the mutable state, stupid.*
2. *Make fields final unless they need to be mutable.*
3. *Immutable objects are automatically thread-safe.*
4. *Encapsulation makes it practical to manage the complexity.*
5. *Guard each mutable variable with a lock.*
6. *Guard all variables in an invariant with the same lock.*
7. *Hold locks for the duration of compound actions.*
8. *A program that accesses a mutable variable from multiple threads without synchronization is a broken program.*
9. *Don't rely on clever reasoning about why you don't need to synchronize.*
10. *Include thread safety in the design process—or explicitly document that your class is not thread-safe.*
11. *Document your synchronization policy.*

Almost all of these rules relate to the difficulties and complexities associated with the synchronization of access to mutable variables and data structures. If mutation is eliminated, most of these problems disappear. Functional programming disallows (or at least discourages) mutation, thus simplifying the development of concurrent programs. Erlang is a functional programming language. When it comes to programming concurrent applications, the consequences of having non-mutable state are enormous. In Erlang, there is no mutable state, there is no shared memory, and there are no locks. This makes it relatively easy to parallelize programs.

3. ERLANG OVERVIEW

Erlang is a general-purpose programming language and runtime environment. Erlang has built-in support for concurrency, distribution and fault tolerance. Named after the Danish mathematician Agner Krarup Erlang, it was developed at the Swedish telecom company Ericsson in the late 1980s. It started as a platform for developing soft real-time software for managing phone switches. It has since been open-sourced and ported to several common platforms. Erlang's minimal set of concurrency primitives, together with its rich and well-used libraries, give guidance to anyone trying to design a concurrent program [10].
In recent years several trade books covering Erlang have been released, including [3, 5, 11]. All of them are geared towards experienced programmers.

3.1 Main Features

Erlang offers some quite interesting features. Here are some of the most relevant ones [6]:

- The language source code is compiled into a platform independent bytecode, which is later executed by a virtual machine (VM).
- A read-eval-print loop (REPL) is provided to allow interactive development and exploration.
- Variables can be bound to a value only once. This is known as "single assignment", and it eliminates several kinds of common concurrency bugs.
- Pattern matching is used to bind variables to values. It can be explicit, as when the `=` (pattern matching operator) is used, or implicit, as when a function call is processed and a function's actual arguments are matched with its formal parameters.
- The type system is strong and dynamic. Variables are not declared. Types are determined at runtime, as is the viability of any operation.

[2]Most common multiprocessor systems today use a symmetric multiprocessing (SMP) architecture. An SMP machine has two or more identical CPUs that are connected to a single shared memory. These CPUs may be on a single multicore chip, spread across several chips, or a combination of both.

- Supported data types include: numbers (integers and floats), atoms, lists, tuples, functions, binaries (untyped arrays), processes, and ports (I/O streams).
- Concurrency is achieved through lightweight processes instead of threads. Unlike threads, processes do not share memory. A typical application can be made from thousands to millions of these extremely lightweight processes. Concurrency belongs to the VM and not the operating system.
- Processes interact only by sending asynchronous messages to each other.
- Location transparency allows sending messages to a process on a local or remote VM using the same semantics.
- Symmetric multiprocessing (SMP) support provides multiple native threads that run Erlang processes. The use of native threads allows machines with multiple CPUs to make effective use of the additional CPUs.
- Garbage collection (GC) is done per process. A system may have thousands of processes, but if GC is only required for a single process, then the collector only touches the heap belonging to that process, thus making collection time practically imperceptible.
- Hot code swapping allows upgrading code without needing to restart an application. This is extremely important for systems that can't tolerate any downtime.
- The standard Erlang distribution comes with the OTP (Open Telecom Platform) framework. OTP is a set of libraries and procedures used for building large-scale, fault-tolerant, distributed applications.
- There are several mechanisms for inter-language working. This means that Erlang code can be combined to work with Java, C++ and quite a few other languages.

Because of its performance characteristics and language and library support, Erlang is particularly good for [10]:

- Irregular concurrency—applications that need to derive parallelism from disparate concurrent tasks.
- Network servers.
- Distributed systems.
- Parallel databases.
- GUIs and other interactive programs.
- Monitoring, control, and testing tools.

On the other hand, Erlang tends not to be well suited for:

- Floating-point-intensive code.
- Code requiring an aggressive optimizing compiler.
- Code requiring non-portable instructions.
- Projects to implement libraries that must run under other execution environments, such as the Java Virtual Machine (JVM) or the Common Language Runtime (CLR).

3.2 Examples

The following examples should give some idea of what Erlang code looks like.

3.2.1 Factorial

Given the mathematical definition for computing the factorial of a positive number n:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

This is how it translates into Erlang code:

```
fact(0) -> 1;
fact(N) when N > 0 -> N * fact(N-1).
```

The `fact` function is implemented by two clauses, separated by a semicolon. The final clause always ends with a period. When `fact` is called, the actual parameters are tested against the patterns in each clause head in turn to find the first match, then the body (after the arrow `->`) is evaluated. Variables, including parameters, are identifiers that must start with a capital letter. The expression after the `when` keyword is a *guard* (a condition that places an additional constraint to the function clause). The value of the final expression in the body is the return value of the call; no explicit `return` statement is needed. As mentioned before, Erlang is dynamically typed, so a call to `fact("boo-boo")` will compile but will raise a runtime exception when an operation with incorrect argument types is attempted.

3.2.2 List Insertion

The following code implements the function `insert`. It takes two arguments: a number N and a list of numbers L in ascending order. It returns a new list with the same elements as L but with N inserted in such a place so as to maintain all the elements of the list in ascending order.

```
insert(N,[]) -> [N];
insert(N,[H|T]) when N < H -> [N,H|T];
insert(N,[H|T]) -> [H|insert(N,T)].
```

The first clause says that if you try to insert a number N into an empty list, the result is a one element list containing N. The second clause says that if you try to insert N into a list, where H is the head (first element) of the list and T is the tail (rest) of the list, and $N < H$, then the result is the list T but with N and H placed at the beginning. If these two clauses do not match (in other words, the list is not empty and $N \geq H$), then the body of the last clause calls itself recursively, inserting N into T; the final result is the list returned by the recursive call but adding H as its head.

3.2.3 Sorting

Using the `insert` function from the previous example, an insertion sort algorithm can be implemented in a fairly straightforward fashion:

```
sort([]) -> [];
sort([H|T]) -> insert(H,sort(T)).
```

The first clause establishes that an empty list is an ordered list. The second clause says that to sort a list, just insert the head of the list into the result of sorting its tail.

3.2.4 Process Handling

The following contrived example demonstrates how to use Erlang's concurrency primitives. Figure 1 is a UML sequence diagram [14] that captures the behavior of what the code does.

```
proc_example() ->
    Proc = spawn(fun () -> do_it() end),
    Proc ! {compute,5,self()},
```

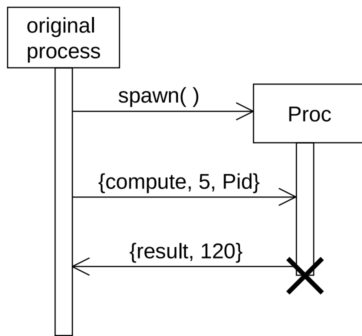

Figure 1: Creating a process, and sending and receiving messages between processes.

```
        receive
            {result,R} -> R
        end.

    do_it() ->
        receive
            {compute,N,Pid} ->
                Pid ! {result,fact(N)}
        end.
```

The `spawn` built-in function creates a new process, returning its process identifier (pid) to the caller. An Erlang process, like a thread, is an independently scheduled sequential activity with its own call stack, but like an operating-system process, it shares no data with other processes—processes interact only by sending messages to each other. The `self` built-in function returns the pid of the caller. A pid is required in order to send messages to a process.

The new process starts executing the anonymous function (defined using the `fun` keyword) specified in `spawn` and will terminate when that function returns.

Messages are sent with the syntax: *Pid* `!` *Message*. A message can be any Erlang value, and it is sent atomically and immutably. It is conventional to use tuples[3] containing atoms[4] as messages. The message is placed in the receiving process's mailbox, and the sender continues to execute—it does not wait for the receiving process to retrieve the message.

A process uses the `receive` construct to extract messages from its mailbox. It specifies a set of patterns and associated handler code and scans the mailbox looking for the first message that matches any of the patterns, blocking if no such message is found. This is the only blocking primitive in Erlang. Like the patterns in function clauses, the patterns in `receive` options match structures and bind new variables. If a pattern uses a variable that has already been bound to a value, then matching the pattern requires a match with that value [10].

In this code, the `proc_example` function creates a new process (bound to variable `Proc`), then sends it the request message `{compute,5,self()}`, and finally waits to receive the response message. The new process starts executing the `do_it` function, which blocks until it receives the expected message. Once this occurs, it sends the message `{result,120}` back to the original sender (referred by the `Pid` variable which got bound to the process identifier returned by the call to `self()` within the original process). The value 120 is the result of computing `fact(N)`, where `N=5`.

3.2.5 Process Distribution

Creating a process so that it runs on another network node is also pretty straightforward. There is a special version of `spawn` that takes the name of the node where the new process will be created and run. Additionally, you must provide the module, name, and argument list of the function to be executed. The distributed version of the `proc_example` function from the last section would be as follows:

```
dist_example() ->
    Proc = spawn(mynode@example.com,
                 mymodule, do_it, []),
    Proc ! {compute, 5, self()},
    receive
        {result, R} -> R
    end.
```

Thanks to location transparency, sending and receiving messages is done just as if all involved processes were running in the local VM.

3.3 Design Principles

In order to achieve a good level of parallelism on a SMP system, Joe Armstrong gives a few recommendations when programming with Erlang [3]:

1. **Use lots of processes in relation to the number of cores.** This will keep the CPU busy. Preferably all the processes should do similar amounts of work.
2. **Avoid side effects.** Side effects are mainly produced by mutating shared memory. A disaster might happen if two threads write to common memory at the same time. Systems with shared memory concurrency prevent this by locking the shared memory while the memory is being written to. The main problem with shared memory is that one thread can corrupt the memory used by another thread, even if the code in the first thread is correct.
3. **Avoid sequential bottlenecks.** This follows Amdahl's law: *The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program* [1]. Sometimes certain steps of an algorithm have to be done sequentially and not concurrently. If that is the case, the only solution might be to replace or rewrite the algorithm.
4. **Write "small messages, big computations" code.** The overhead of setting up a process and waiting for a reply should not be greater than the time required by the process to do the computation itself.

3.4 Drawbacks and Alternatives

Erlang is not the silver bullet for concurrent programming. Many issues still arise as noted in [10]:

[3] Erlang tuples are a compound data type with a fixed number of terms. They are delimited with curly braces: `{`$\mathtt{Term}_1$`, `$\mathtt{Term}_2$`, ..., `$\mathtt{Term}_n$`}`.

[4] An atom is a literal, a constant with a name. They are equivalent to symbols in languages like Lisp or Ruby. They must start with a lowercase letter.

> All standard errors in concurrent programming have their equivalents in Erlang: races, deadlock, livelock, starvation, and so on. Even with the help Erlang provides, concurrent programming is far from easy, and the nondeterminism of concurrency means that it is always difficult to know when the last bug has been removed.

There are alternative concurrency mechanisms supported by other modern programming languages; some of these might be considered simpler than Erlang's message-passing model, specifically in the context of non-distributed applications. An interesting example worth mentioning is the fairly new Clojure programming language [9]. Clojure is a non-pure functional Lisp dialect that runs over the JVM. It supports threads and shared memory, but it protects the later via software transactional memory (STM). STM is a higher-level approach to thread safety than the locking mechanisms provided by Java and other mainstream languages. Rather than creating fragile, error-prone locking strategies, shared state can be protected with transactions, which are easier to reason about.

4. CLASS EXPERIENCE

In August 2007, the author introduced Erlang and COP in his *Programming Languages* course at the Tecnológico de Monterrey, Campus Estado de México. This course is part of a nine semester Computer Systems Engineering undergraduate program, which basically blends together computer science with software engineering.
The author has been teaching this course since 1994, using several different languages at different moments. The languages that have been used in the past include: Scheme, Prolog, Python and Ruby.

4.1 Syllabus

Students usually take the *Programming Languages* course during their fifth or sixth semester. Its prerequisites include *Data Structures* and *Theory of Computation*. The course syllabus, including the approximate number of classroom hours assigned to each topic, is as follows:

1. General Concepts of Programming Languages (3 hrs.)
2. Functional programming (15 hrs.)
3. Parallel programming (12 hrs.)
4. Distributed programming (9 hrs.)
5. Introduction to Language Translation (9 hrs.)

Erlang is used from topics 2 to 5. At the beginning of the course, students have no previous exposure to Erlang, nor to functional programming for that matter. They start learning the sequential part of Erlang during topic 2. The larger portion of the course, topics 3 and 4, is dedicated to COP.
A semester is comprised of 16 weeks, with three classroom hours per week. About 80% of the classroom time is spent in lectures; the remaining 20% is used for quizzes, exams, and a few collaborative activities. Students are expected to schedule an additional five hours a week for extra class activities on their own.
The course final grade is calculated considering the following items: programming assignments 10%, final project 10%, discussion forums 5%, quizzes 15%, quarter term exams 35%, and a final exam 25%. A student requires a final grade of least 70 out of 100 in order to pass the course.

References [5, 15] are used as textbooks. These books were chosen not only due to the adequacy of their content, but also because students have access to them through the *Safari Books Online* campus subscription [12].

4.2 Extra Class Assignments

During the semester, students are required to use Erlang to develop individually five programming assignments and a final project. A programming assignment consists of three to ten problems. Each problem is solved usually in just a few lines of code. Students are given at least one week to solve each programming assignment.
This is the list of programming assignments:

1. **Recursive Functions.** Students solve problems that require a recursive solution and pattern matching. For example: Write a function that, given a list of elements, returns a new list that results from compressing the original list using the run-length encoding (RLE) algorithm.
2. **Higher-Order Functions.** Students solve problems that require using higher-order functions[5]. For example: Implement functions that allow to do numerical derivation and integration.
3. **Project Euler.** Students select and solve at least ten problems from the *Project Euler* [13] website, which contains a series of challenging mathematical problems of different difficulty levels that can be solved by writing computer programs in any language.
4. **Concurrent Programming.** Students solve problems that involve creating processes, and sending and receiving messages. For example: create a ring of N processes, and send a message M times around all the processes in the ring.
5. **MapReduce Exercises.** Students solve problems that use the MapReduce framework [8] in order to run efficiently on an SMP system. For example: Write a function that determines the IP address of the remote host that has the greatest number of accesses to a certain web server by inspecting the corresponding log files. Students also perform benchmarks to evaluate the performance of their solutions.

The first three programming assignments allow students to develop functional programming skills, while the last two focus on parallel programming abilities.
For the final project, students build a complete text based distributed multi-player game, working in teams of two or three people. This is a fairly challenging activity. A different game is requested each semester. So far, these are the games that have been implemented by students: Tic-Tac-Toe, Connect-4, Dots and Boxes, Crazy Eights, Go Fish!, and Collector. To successfully complete the project, the team members must effectively write functional, parallel and distributed code in Erlang, thus demonstrating a good level of COP proficiency.

4.3 Results Obtained

In the last six semesters, 166 students have taken the *Programming Languages* course with the COP approach using Erlang. 87.35% of those students obtained a passing grade.

[5]A higher-order function takes one or more functions as an input, and/or returns a function as its ouput.

A passing grade guarantees that a student has at least the basic skills needed to write code using the functional, parallel and distributed programming styles.
In order to have a better idea on how students perceive the material covered in the course, the author conducted in the last two semesters a fairly simple anonymous survey. At the end of the course, students were requested to honestly give a score to six statements depending on how closely they resembled their current attitudes and feelings. The possible score values were: 5 ← Strongly Agree, 4 ← Agree, 3 ← Neutral, 2 ← Disagree, and 1 ← Strongly Disagree. These were the statements to be scored:

1. I consider Erlang's concurrency message-passing model to be easier to understand and use than the model supported by C/C++/C#/Java, which uses threads, shared memory and locks.
2. I consider that writing programs that fully take advantage of the parallel facilities offered by multi-core systems is easier to achieve using Erlang than using languages like C, C++, C# or Java.
3. I consider **functional programming** to be easier **in general** than object oriented or procedural programming.
4. I consider that **functional programming** simplifies the development of **concurrent** programs, compared to object oriented or procedural programming.
5. I consider that **functional programming** simplifies the development of **distributed** programs, compared to object oriented or procedural programming.
6. If I had to build a **concurrent** or **distributed** system from scratch, I would most definitely consider using Erlang.

A total of 44 students took the survey. Of those, 84% had already taken an *Operating Systems* course. This is relevant because that course also covers concurrency, but with the traditional model using threads, shared memory and locks. 100% the students also mentioned that they already had some experience using threads in Java, C or C++ from some previous course.
The following table shows the results of the survey. Note that $\overline{x}$ is the arithmetic mean and σ is the standard deviation of the entire population.

Stmt. No.	$\overline{x}$	σ
1	4.34	0.88
2	4.45	0.58
3	3.20	1.14
4	4.34	0.67
5	4.25	0.83
6	4.07	0.75

5. CONCLUSIONS

From section 4.3, it is interesting to note that most students "Strongly Agree" or "Agree" on all statements except for number 3, which also happens to have the largest deviation. It seems that students consider Erlang and functional programming to be good options for COP, but there is disagreement on the general relevancy of functional programming.
When students start using Erlang, it appears to be quite odd to them. But after using it for a while, they start to appreciate its merits. When the semester ends, most students seem to enjoy using it.
Erlang is definitely not a perfect language. But, in the author's experience, Erlang is better suited for teaching concurrency than the alternatives that most instructors are currently using.
The material used for teaching COP and Erlang is available under a Creative Commons Attribution-Noncommercial license at the following URL: *http://www.arielortiz.com/*

6. REFERENCES

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings*, (30):483–485, 1967.
[2] J. Armstrong. *Concurrency Oriented Programming in Erlang*. http://www.guug.de/veranstaltungen/ffg2003 /papers/ffg2003-armstrong.pdf Accessed December 6, 2010.
[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
[4] Association for Computing Machinery and IEEE Computer Society. *Computer Science Curriculum 2008: An Interim Revision of CS 2001*. http://www.acm.org/education/curricula /ComputerScience2008.pdf Accessed December 6, 2010.
[5] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.
[6] Ericsson Computer Science Laboratory. *Erlang Programming Language, Official Website*. http://www.erlang.org/ Accessed December 6, 2010.
[7] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.
[8] Google Code University. *Introduction to Parallel Programming and MapReduce*. http://code.google.com/edu/parallel /mapreduce-tutorial.html Accessed December 6, 2010.
[9] R. Hickey. *Clojure Website*. http://clojure.org/ Accessed December 6, 2010.
[10] J. Larson. Erlang for concurrent programming. *Queue*, 6(5):18–23, 2008.
[11] M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning Publications, 2010.
[12] O'Reilly. *Safari Books Online*. http://my.safaribooksonline.com/ Accessed December 6, 2010.
[13] Project Euler. *Project Euler Website*. http://projecteuler.net/ Accessed December 6, 2010.
[14] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley Professional, 2005.
[15] M. Scott. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann, 2009.
[16] H. Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. http://www.gotw.ca/publications /concurrency-ddj.htm Accessed December 6, 2010.