On Language Support for Application Scalability

Patrick Bader

November 2, 2009

mail@patrickbader.eu

Contents

1 Introduction

In the past years increasingly large systems, mainly internet sites, were built. This is due to the enormous growth of the web 2.0 with its social communities like facebook and video platforms. The requirements of these systems can be categorized in two groups: software and hardware requirements. While computer hardware is evolving rapidly, according to Moore's Law, there is a lack in software enhancements. That means we do have the hardware to build large systems but it's extremely hard to write software for them.
Thus this paper will address the software part, especially the part of programming languages. Choosing the right programming language(s) often decides on the success of a system, so in this paper, some programming language requirements will be shown and will be discussed afterwards.

2 Large-scale System Requirements

When thinking of large-scale systems, social communities like facebook or video hosting plattforms like YouTube come in mind. These systems are used by millions of users at the same time and produce remarkably heavy traffic on the internet[1]. On the technical site are many servers distributed all over the world to handle requests from millions of users.
What does not come in mind is these systems did not start big, most of them were run by startup companies which did not have funds for dozens of servers. Something that started small may eventually grow exponentially within a short period of time. This is often a problem for todays way of programming. Sometimes changes to the system have to be made within days or even hours[1]. Alan Kay illustrates the problem when he says:

> I was thinking about ecological computing. When I was working with computers in the late '60s, all of the computer power on Earth could fit into a bacterium. The bacterium is only 1/500th of a mammalian cell, and we have 10 trillion of those cells in our bodies. Nothing that we have fashioned directly is even close to that in power. Pretty soon we're going to have to grow software, and we should start learning how to do that. We should have software that won't break when something is wrong with it. As a friend of mine once said, if you try to make a Boeing 747 six inches longer, you have a problem; but a baby gets six inches longer ten or more times during its life, and you never have to take it down for maintenance.

Some fundamental programming languages requirements, being the basic tools for software development, can be extracted from the description above.
The human body consists of trillions of cells interacting with each other simultaneously which ultimately leads to its immense power. Programming languages should therefore provide means of doing different things at the same time. This mechanism is called parallel programming and discussed in the following chapter.
The larger a system is, the more likely faults occur. In most of today's software a lot of work is invested for exception handling and error avoidance, but if a fault occurs the whole system will crash. Since a system crash for an organism like the human body

means death, it has to be highly fault tolerant. That means it has to handle wrong behaviour or defects caused by e.g. viruses or bacteria it has never seen before.
Lastly human beings grow, they evolve from merly a single cell to what is one of the most complex systems known. The question here is, how can such a growth be applied to programming? Living beings are growing, that means a system should be running when it grows. The majority of common programming languages does not support code changes in a running application.
Lastly complexity of large systems has to be faced in some manner. The last chapter will thus concentrate on improvements for programming languages in general, like syntax.
Putting it all together programming language requirements for scalable applications are:

- Parallel programming
- Fault tolerance
- System updates
- Syntax and semantics

3 Parallel Programming

Already living in a multicore world, programmers have to face the challenge of parallel programming to utilize all available CPU power. Programming with concurrency in mind adds a lot of complexity since the human mind works sequentially which makes thinking in concurrent tasks hard.
In this chapter some fundamentally different approaches to parallel programming will be discussed. First the most common approach with threads and shared memory. Languages that support this model are Java or C# for example.
The second approach uses the actor model, having no shared memory but asynchronous message passing at language level. An example of such a language is Erlang, a functional language that has been used successfully in highly available systems.

3.1 Multithreading

Multithreading comes from the need of having very lightweight processes that can access data with minimal overhead. Operating system history shows that this concept is known from the beginning. The first operating systems that supported multiprogramming had one address space which was shared by the running programs. This lead to serious problems including security issues and dependencies of unrelated programs.
The result was a separatation of the address space which wiped away most of the problems with a shared one. The solution is at the cost of performance in inter process communication. So threads were built as a step back to the old way of multiprogramming with all its downsides.
Most of the downsides however can be avoided in some cases. There are applications especially with a large part of data parallelism like most of image processing or rendering.

Large-scale systems like social community web sites demand for task parallelism instead of data parallelism. This in mind, programming with threads becomes difficult. Extensive locking is required to keep data in a consistent state and interaction of many threads becomes unmanagable. Understandable sequential program flow becomes a complex situation in which humans regularly fail. Dietrich Dörner shows in 'Die Logik des Mißlingens'[2] that few people are able to manage systems in which many parts concurrently influence one another.
When moving from data to task parallelism another technical problem arises: In mainly data parallel applications few threads are running, approximately as much threads as there are hardware threads available (depending on the number of CPUs and cores). In a task parallel application, for example a web server, an intuitive approach would be to have as much threads as there are users. Having 10.000 users at the same time and as much kernel threads[1] will result in a system with nearly zero throughput. Context switches of kernel threads are quite expensive and thus are a limiting factor to the overall thread count. Alternatively lightweight user level threads can be used to solve the context switching problem. The disadvantage is the operating system knows nothing about these threads and thus cannot distribute work on the available CPUs.

3.2 Functional Programming and Actor Model

Functional programming, especially its first implementation LISP[3], is known for almost half a century by now and was nearly forgotten since the rise of object oriented programming. Recently functional programming has its renaissance in languages like Erlang, F# or Scala. Even C#, an object oriented language, has limited support for functional programming in the shape of lambda expressions for example.
There are two reasons for this development:

- Many algorithms can be described in a functional style with few lines of code without the overhead of defining classes to wrap the code. Thus raising productivity and clearness of code.
- Pure functional programming is side-effect free. That means whenever a function is called with some arguments, if the arguments are the same, the return value of the function will be the same. There is no such thing as shared state that can be modified by a function.

Side-effect free functions can easily be called in parallel without having to lock resources. This is taken one step further by the actor model. Each actor encapsulates some functionality and is independend of other actors. Actors communicate through asynchronous message passing only, which differentiates them from objects in the object oriented world. Actors can perform tasks simultanously. Joe Armstrong, one of the inventors of Erlang, thus calls this approach to programming Concurrency Oriented Programming [4].
In Erlang, each actor has its own thread of execution. These threads are called processes since their nature of not sharing data and their similarity to operating system

[1] Threads that are managed by the operating system

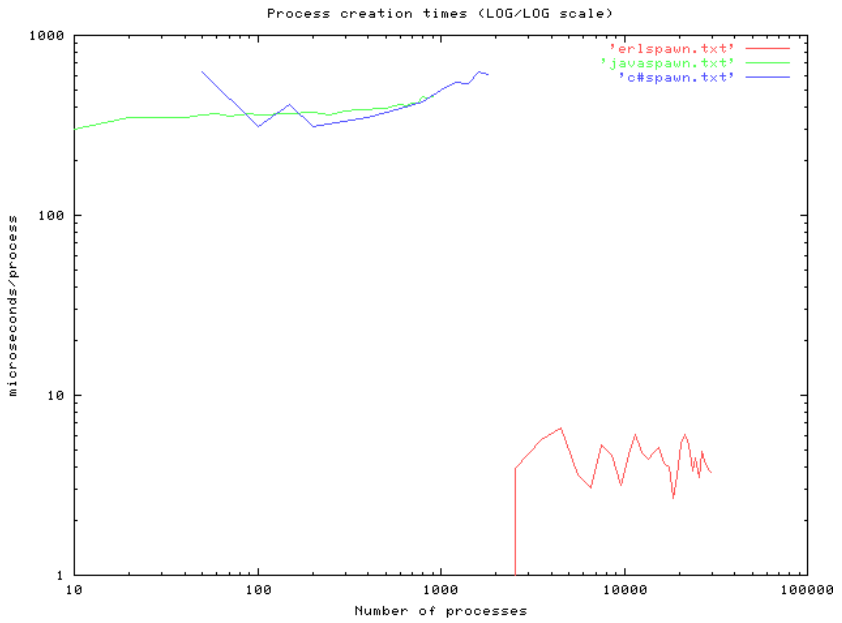

Figure 1: Process creation times. red : Erlang, green : Java, blue : C# source: [5]

processes. In contrast to processes on operating system level, Erlang processes are extremely lightweight, shown in figure 1. They can easily be spawned and a single machine is able to manage several hundred thousands of these processes.
Internally Erlang uses both user level threads to achieve high process counts and kernel threads to utilize hardware threading. So the main difference between this approach and the previous one is threads are abstracted to a higher level construct which is supported on language level. The goal is to hide concurrency as good as possible from the programmer. Erlang allows server programming with mostly sequential code through so called behaviours that are high level building blocks for different kind of servers. More detailed information can be found on the Erlang website: http://www.erlang.org or in Joe Armstrong's book[4].
The idea of language support for concurrent programming can also be taken one step further to distributed programming. There, some new Problems arise, especially in the area of software or hardware faults. Thus, the next section will cover the topic: Fault tolerance.

4 Fault tolerance

In large systems, especially distributed systems, many different fault types can occur. This section will show the most common ones, how they can be indicated and what their influence on programming is. Since most programming languages do not have the ability to detect faults, detection and recovery has to be built into application code, which adds another aspect to application development and thus raises complexity. Using the example of Erlang, which was also designed for distribution, some concepts for handling or at least detecting faults on language level will be shown.

4.1 Fault types

Basically two types of faults in distributed systems occur. In the first category are hardware faults, like server crashes or network problems. In the second category are software errors, which may occur due to communication protocol violations, for example, or simply programming errors.
To recover from faults or at least detect their source is hard, especially in a multithreaded or distributed system. One basic building block required for fault recovery is exception handling. Despite a variety of programming languages have some kind of exception handling construct, support for handling them in threaded or distributed environments is rather limited.
Whereas the current C++ standard for example does not mentioned anything about multithreading at all and thus does not provide any means of fault recovery beyond threads[2], newer languages like Java do provide methods[3] but leave much work to the programmer in this case, so they are of limited use.

4.2 Exception domains

In which case should exceptions be handled? An Exception has to be handled if the specification defines a way to deal with it, else some kind of error occured that cannot be handled. For example when a user tries to open a file that does not exist in a texteditor, the specification could define that an message to inform the user will show up, else an error occured.
What happens if an exception is not handled at all or some other kind of fault occurs? Three different outcomes are possible and shown in figure 2. The initial fault is marked by the red cross in part a), which means that one process[4] fails, probably caused by an unhandled exception.

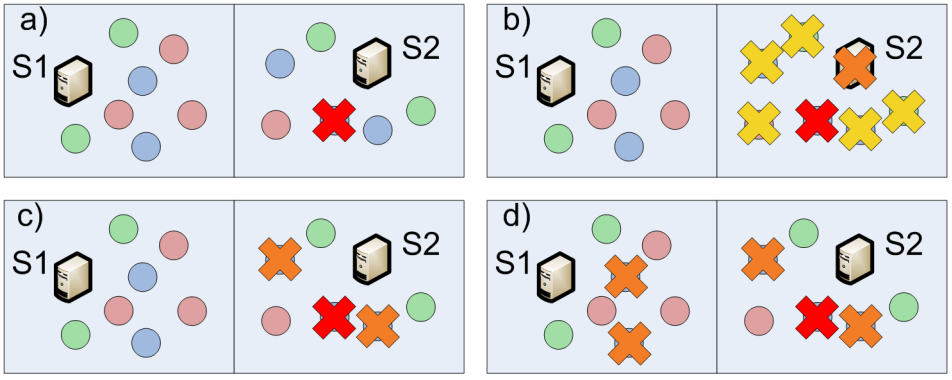

Figure 2: Different outcomes of unhandled exceptions. Circles represent different threads or processes each belonging to one server (S1 or S2) illustrated by the two boxes. Circles with the same color are depending on each other. Part a) shows the source of the fault.

The behaviour of most languages is shown in part b): A fault in one thread causes the whole application to crash, even if there are other independent subsystems. This is fatal

[2]Most compilers like the GNU, Microsoft or Intel compiler however allow multithreaded programming.
[3]The mechanism is explained in the Java API[6]
[4]The words 'process' and 'thread' are treated equivalently here and in the following sections.

for building fault tolerant systems, since system stability depends on the stability of the weakest subsystem.
Part c) shows a better approach, only the subsystem that fails will be shut down, the rest of the application is untouched. Having the ability to restrict the crash on a specific subsystem increases the overall system stability but leaves some orphaned ones on other machines running.
Part d) goes even one step further. Whereas only processes that are on the same machine as the crashed one are being killed in the previous part, in this, even processes on other machines that rely on the crashed one, will be killed. This leaves no processes that depend on killed ones running and also guaranties that unrelated ones stay unaffected. Erlang as an example supports this model and allows exact control over the fault recovery process with the help of supervision trees. The basic principles of supervision trees are briefly described below, for a more complete explanation, especially in case of the Erlang implementation reading [7] is advised.

4.3 Supervision trees

Erlang programs consist of many interacting processes as already statet in section 3.2. Each process can be linked to every other process. Linking two processes means both depend on one another. Linking is biderectional, if one of two processes that are linked together dies the other will follow. This mechanism is independent of the location of the process and equally works on remote ones. So linked processes fulfill the requirement of figure 2 d). The Erlang VM abstracts unreliable software and hardware. If one machine fails because of a failure in the power supply, all linked processes will be killed by default.
Fault tolerance not only means letting parts of the system crash without others being affected, it also implies availability. Since crashed processes are not available, they have to be restartet whenever they crash. Some kind of uniderectional link, which in Erlang is called monitor, is needed to fulfill this requirement. If a monitored process crashes the monitor gets noticed and can bootstrap the failed processes according to some rules.
Since a monitor itself is a process it can also be monitored, thus the mechanism of having multiple levels of monitoring is called supervision tree. A prerequisite of supervision is processes (or threads) have to die if something goes wrong which contradicts the Java approach for example. Introducing checked exceptions forces the programmer to handle situations inside a thread that simply can't be handled in a reasonable way. This requirement is in depth discussed in [7] in chapter 2.10.
Monitors can also be used remotely which grants the ability to recover from machine crashes for example. If a monitor were local, it would fail at the moment the machine crashes, too. With remote monitors the killed processes can be respawned on a working machine.
Yet, spawning a process on different machines introduces another issue. New code may have to be distributed to a different machine at runtime because it might not have loaded the code that was running on the crashed server. The topic of software updates and especially hot code swapping, to solve this issue, is therefore discussed in the following section.

5 Updating Software without Downtimes

One main goal of large-scale systems is high availability. One cannot imagine a single day with the google search site down for maintenance, for example. But how about bug fixing and system updates? What is a system worth that gets bugs fixed in a very small period of time, but is hardly available since the system has to be taken offline each time a patch is applied? This section will show concepts of how availability can be preserved in the presence of code change.

5.1 Hot Code Swapping

Figure 3 a) shows a classical programming cycle. For simplicity reasons testing and other factors are left out. The problem with this approach is as follows. When some code is changed or added, after the compilation phase, the test system has to be startet, meaning changing some binaries, running the application, and proceeding to the altered program section. After a bug has been found the system has to go offline and programming is continued. Although these steps can be automated to some degree they consume quite a lot of time the programmer can do nothing but wait.

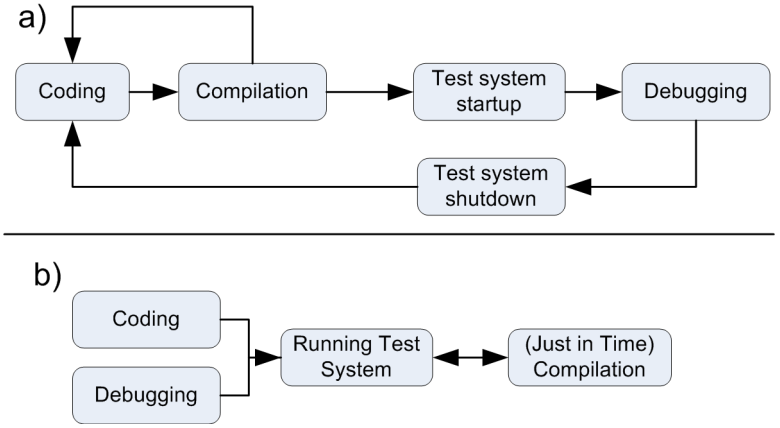

Figure 3: Two different programming cycles. a) classical programming b) programming in a live system

In the past some effort has been made to overcome this problem. An example is Smalltalk, in which the programmer operates in the running system. There is no need for restarting and changes can be applied on the fly or even while the debugger is running. This approach increases productivity by reducing the waiting period and is shown in figure 3 b). The mechanism of code change in a running system is called hot code swapping. Whereas it is possible to write an update on an application server with thousands of users on it, a programmer would not want to do it.
Why will the mechanism still be an advantage for high availability, if it is used on test systems only? The answer is simple, the same mechanism cannot solely be used while some update or patch is being developed on a test system, it can also be used for deploying an update to a live system. This has to be done in two steps: distributing the code and finally updating running code. The split is necessary to apply updates as fast as possible.

Applying large updates on running systems is a topic of its own since there are quite a few pitfalls concerning dependencies between processes etc. and is thus out of the scope of this paper. For example some kind of versioning has to be made in the updating process in which some old and some new code is running at the same time.

5.2 Hot Code Loading

Another advantage is shown in distributed systems. As stated in the previous section failure some server has to take over on server failure. But this server does not have to have the necessary code loaded, so every server with little load will do. With hot code swapping or in this case, hot code loading, the new server is able to load needed code on demand. This saves resources and keeps the system flexible. Joe Armstrong describes the principle of a generic server which can take over any task in his Erlang book[4].
As already indicated, Erlang supports hot code swapping and proofs this mechanism truely working. Erlang is used in highly available[5], carrier-grade ATM switches produced by Ericsson and various other projects[6], notably the facebook chat backend is implemented in Erlang, too.
As seen before, hot code swapping increases productivity by reducing the time programmers have to wait until code changes can be tested and debugged. The next section will cover other methods to increase productivity.

6 Syntax and Semantics

In the previous sections, some of the most important programming language requirements to help building large-scale systems have been shown. The question to ask next is, what makes a system large-scale? The obvious answer would be, systems with many lines of code are large-scale. Is this really true? Surprisingly there is a famous quote from Bill Gates about this topic, saying:

> Measuring programming progress by lines of code is like measuring aircraft building progress by weight.

So we really do not want to write lots of code but what is the gain of writing less?
According to [8] and [9], the amount of errors in code is proportional to the number of lines of code and the amount of code which can be written in a specific time is constant. This means, if a program can be written with only half as many lines of code, there will be half as many errors and coding will be finished twice as fast. So system size should better be measured in user or feature count instead of lines of code.

6.1 Moore's Law

According to Moore's Law, transistor count on a single CPU has been increasing exponentially, which can be seen in figure 4. This development makes increasingly large systems technically possible. However instead of only system size growing, the lines of

[5] The availability of the AXD301 ATM switch is reportet to be nine nines.
[6] A list can be found in the Erlang FAQ at http://www.erlang.org/faq/faq.html#AEN50.

code count for these systems were increasing similarly, which can be seen exemplary in the development of the linux kernel, figure 5. The result is hardly maintainable code with increasing numbers of bugs and thus unstable systems.

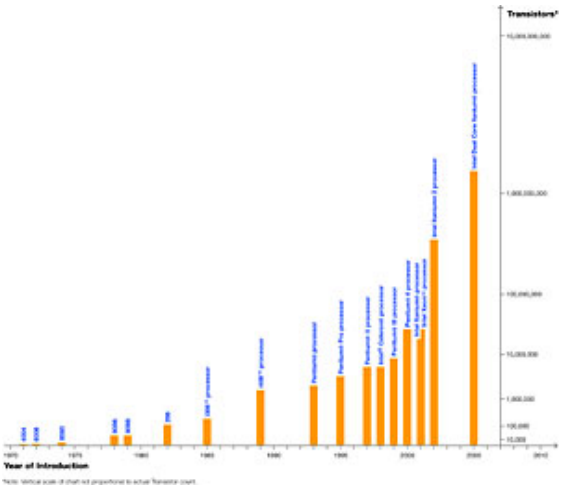

Figure 4: Microprocessor development over time. Source: [10]

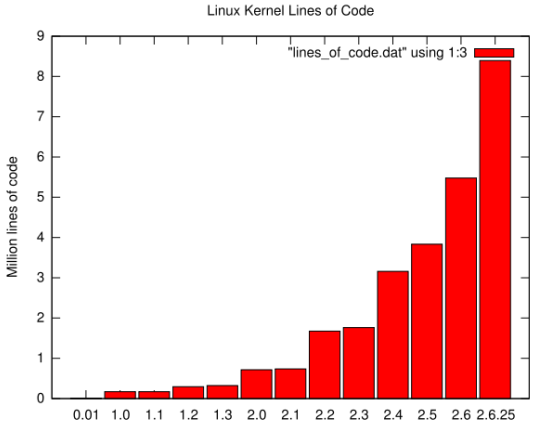

Figure 5: Growth of the Linux kernel. Source: [11]

What can be done about it? The easiest and shortest term solution is simply to use the right tools for a problem. This general rule also applies to programming languages and will be discussed shortly. Another solution would be to use very high level languages or even use extendable languages.

6.2 Using the right Tools

One should use the right tools to do something efficiently. Though this sentence sounds simple and no one would ever try to use a spoon for cutting something, software developers seem to do so. There currently is no such thing as one language for everything. Whereas Java for example, once developed for household machines, now gets used for enterprise applications. Similarly XML is used both for documentation and object persistance. The result is, more and more technology and specifications have to be developed to workaround language deficiencies.

For example XML documents may be logically equivalent but not physical[7]. So there is a standard to preserve physical equivalence at the cost of document readability. So to check for integrity of an XML document, it has to be parsed first and transformed to form specified for binary compatibility. Even binary or computer generated data like IDs are serialized. One of the arguments for XML was, it is both machine and human readable. The result is: One technology gets abused for the strangest applications.
This is definitely not the way to do things right. Technology should be used for the things it was developed for and not because everyone uses it or one does not want to learn another one. The people at facebook have understood that very well. Facebook uses a dozen of different technologies for example Java, C++, Ruby, Haskell, Perl, Erlang,...
The disadvantage of using different languages at the same time is, some glue code is necessary to connect modules written in one language with modules of another. A way out of the problem is tried by Microsoft and Sun, for example. Both have built a virtual machine with support for several different programming languages on top of it. Thus plugging code together is much easier.
Microsoft for example added a functional language called F# to the .NET framework since some problems, especially many algorithms are more elegantly expressed in a functional style. On Sun's side is Scala, also functional and increasingly popular. Thus it is on the software developers to use available technologies and the management to let them.

6.3 Very High-Level Languages

A somewhat different approach is the use of very highlevel languages. These are languages with special syntactical constructs that are able to reduce the amount of code to solve a problem significantly. Some research in this field is made by a team around Alan Kay at Viewpoint Research Institute. The project is called "STEPS Toward The Reinvention of Programming" [12].
The goal is to build a whole personal computing experience somewhat similar to what is currently known as the operating system with some basic applications in less than 20000 lines of code and from scratch. But how can it be achieved? The answer lies in the use of very high-level languages that are designed for a specific problem domain. Unlike DSLs[8] are specialized languages for one problem domain the goal of STEPS is to build a whole system, with many different languages, from hardware to the end-user. Heart of the system are facilities to efficiently build new languages. The core is IS a parametric machine code compiler, which written in itself can be expressed by 1000 lines of code. As an example, a working JavaScript can be implemented in about 170 lines of code. Being able to easily develop high-level languages that are compatible with the whole system heavily decreases the overall amount of code. Research on this topic is by far not complete, so whether it will be a success remains to be seen.

[7] Physical equivalence is important for generating and validating hashes which is needed for most security mechanisms.
[8] Domain Specific Language.

6.4 Extensible Languages

Instead of having many languages for specific problems one can think of a language that is able to adapt its syntax and semantics to problem domains. Such a language has to have the ability to change its own grammar on runtime, making grammar a first-class citizen. In fact, in the 1960s and 70s, some programming languages were built that were able to be extended by the programmer, IMP[13] and Lithe[14] being two of them. Though this approach seems to have been abandoned recently, it may have its renaissance in the near future.

7 Conclusion

Current development shows that a change in the way of software development takes place, with web 2.0 applications being one of the pushing forces. More and more features have to be implemented in programming languages that allow for system scalability. Whereas some features are available in relatively new languages like Erlang, others are yet to come. Modular language design seen in research projects like STEPS might get a key principle for upcomming languages.
The trend in software development goes from the one solution for all problems approach to multiple language systems in which each language is seen as a tool made for a specific task. With the ability to change code on runtime and being able to recover from software faults, programs writtten in such languages are able to grow over time. So large-scale systems can grow with less programming effort and thus both higher productivity and reliability.

References

[1] C. Do, “Youtube scalability.” Google TechTalk, June 2007.

[2] D. Dörner, *Die Logik des Mißlingens*. Rowohlt Tb., December 2003.

[3] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine (part i),” *Communications of the ACM*, April 1960.

[4] J. Armstrong, *Programming Erlang Software for a Concurrent World*. The Pragmatic Programmers, 3.0 ed., July 2008.

[5] J. Armstrong, “Concurrency oriented programming in erlang.” http://ll2.ai.mit.edu/armstrong-talk.pdf, November 2002.

[6] I. Sun Microsystems, “Java 2 platform standard ed. 5.0 interface thread.uncaughtexceptionhandler.” http://tinyurl.com/2u8bl6.

[7] J. Armstrong, *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 2003.

[8] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.

[9] U. Wiger, "Four-fold increase in productivity and quality." Workshop on Formal Design of Safety Critical Embedded Systems., March 2001.

[10] Intel, "Moore's law 40th anniversary." http://tinyurl.com/yl6nmgb.

[11] G. Hauser, "Codezeilen des linux kernels," December 2007.

[12] A. Kay, "Steps toward the reinvention of programming first year progress report." http://www.viewpointsresearch.org/html/writings.php, December 2007.

[13] W. Bilofsky, *PDP-10 IMP72 REFERENCE MANUAL IMP72 Version 1.5*, August 1972.

[14] D. Sandberg, "Lithe: a language combining a flexible syntax and classes," in *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 142–145, ACM, 1982.

[15] A. Warth, *Experimenting with Programming Languages*. PhD thesis, University of California, Los Angeles, 2009.