Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Draft, 2013

ProcessJ: A Possible Future of Process-Oriented Design

Jan Bækgaard PEDERSEN [a,1] and Marc L. SMITH [b]

[a] *Department of Computer Science, University of Nevada Las Vegas, NV, USA*
[b] *Computer Science Department, Vassar College, Poughkeepsie, NY, USA*

Abstract. We propose ProcessJ as a new, more contemporary programming language that supports process-oriented design, which raises the level of abstraction and lowers the barrier of entry for parallel and concurrent programming. ProcessJ promises verifiability (e.g., deadlock detection), based on Hoare's CSP model of concurrency, and existing model checkers like FDR. Process-oriented means processes compose, unlike thread-based or asynchronous message-passing models of concurrency; this means that programmers can incrementally define larger and larger concurrent processes without concern for undesirable nondeterminism or unexpected side effects. Processes at their lowest, most granular level are sequential programs; there are no global variables, so no race conditions, and the rules of parallel composition are functional in nature, not imperative, and based on the mathematically sound CSP process algebra. Collectively, these ideas raise the level of abstraction for concurrency; they were successful once before with the occam language and the Transputer. We believe their time has come again, and will not go away, in this new age of multi-core processors. Computers have finally caught up with CSP and process-oriented design. We believe that ProcessJ can be the programming language that provides a bridge from today's languages to tomorrow's concurrent programs. Learning or teaching the programming model and language will be greatly supported through the educational part of the proposed project, which includes course templates and an online teaching tool that integrates in-browser programming with teaching material. Our efforts are encouraged by the forthcoming 2013 IEEE and ACM curricula guidelines, which for the first time include concurrent programming as a core knowledge area at the undergraduate level.

Keywords. ProcessJ, process-oriented programming

Introduction

Over the last decade, a new architecture model has been introduced to the world of computing: multi-core processors. As speeds of conventional single-core processors plateaued, a new approach had to be taken to continue the performance increase so desperately sought by CPU manufacturers and users.

Today, it is not uncommon to find 32-core or even 64-core processors, and there is no reason to believe that this trend will not continue. Although the idea of using multiple CPUs to solve large problems is not new — message passing and shared memory architectures have been around for a long time — the reality of multiple cores on a single socket (chip) sharing resources is. Meanwhile, programming such architectures to make good use of the multiple

[1]Corresponding Author: *Jan Bækgaard Pedersen, University of Nevada Las Vegas, 4505 Maryland Parkway, Las Vegas, NV, 89154, United States of America*. Tel.: +1 702 895 2557; Fax: +1 702 895 2639; E-mail: `matt.pedersen@unlv.edu`.

cores available is difficult, time consuming, and error prone. On a quad-core processor, sequential code cannot use more than 25% of the processing power. If more is needed, there has to be concurrently executable code, and lots of it. For there to be a chance of keeping the cores busy with useful work, there needs to be an order of magnitude more concurrency in the software than is available in the hardware: the principle of parallel slackness, first discussed by Valiant [1].

Asynchronous message-passing programming (e.g., MPI_Send/MPI_Recv), which is typically fairly coarse-grained parallelism, will not make optimal use of the computing resources; multi-threaded programming, which requires locks on shared resources, is a complicated and often dangerous game. Debugging such programs is, at best, hard; even worse, because of possibly rare race conditions, programs that during testing seem fine, might deadlock or livelock if scheduled differently. In addition, trying formally to argue about or prove properties of lock-based programs is rarely achievable.

Despite all these difficulties, concurrency is needed to make full use of computing resources. We believe we have a better model that is more appropriate and much safer. This programming model for multi-core processors does not require locks and semaphores, etc.; it does not require the programmer to code explicitly for N cores; it reduces the number of potential errors associated with the concurrent part of the program by means of a solid foundation in mathematics as well as automatic verification and formal model checking; and it is highly efficient and easily teachable. In this paper we propose a complete suite, from programming language through online tools to IDEs that we believe could serve as a possible way to reintroduce process-oriented programming to the general computing community in the twenty-first century.

1. Existing Programming Models and Languages

Let us briefly point out some of the problems and issues we perceive to exist in current programming models that could be used to program multi-core CPUs. We focus on the threads and locks model, asynchronous message passing, and languages (besides occam [2,3] and occam-π [4,5,6,7,8,9,10]) that are inspired by CSP [11,12,13,14], but have not broken into the mainstream.

1.1. Thread and Lock Model

Shared memory initially seemed like a good idea: treat data as a shared resource between concurrent threads of execution. This model provides a way for threads to communicate with each other. However, this solution creates even greater problems: nondeterminism, race conditions, and the need for mutual exclusion. Without shared memory, these problems don't exist.

Access to shared memory requires the use of locks, semaphores, and mutexes, etc. Such synchronisation primitives are hard to master and get right; reasoning about a program with just a few locks quickly becomes a complicated task, and even very capable programmers get it wrong. For example, a solution to the Santa Claus problem, written in Java with locks [15], was shown to be incorrect [16].

Programs written using the thread-and-lock model are error prone; even if the error does not show up in tests, such programs are hard to reason about or even prove correct. Testing is challenging enough for sequential (deterministic) programs, and its importance is well established. There are even techniques for testing nondeterministic, randomised sequential programs, which include repetition and testing for a range of expected results. However, concurrency presents another layer of complexity due to runtime scheduling-based nondeterminism: race conditions, or even deadlocks, can remain hidden throughout long, extensive periods of system testing, and not reveal themselves until unfavorable scheduling occurs after a system is in production.

Although all cores on a multi-core CPU share the main memory, it is not necessary to use shared memory primitives (explicitly) to program it. We argue that a model with synchronous channel (point-to-point) communication with no shared data is a much safer alternative; actually, it is so safe that we can formally prove that our programs never deadlock or livelock, and race conditions can never happen, ever!

1.2. Asynchronous Message Passing

Coarse-grained parallelism, as found in asynchronous message-passing programming, is not entirely suitable for a multi-core architecture either. Often, such systems are developed with a certain architecture size in question, frequently using a master/worker model as well as the embarrassingly parallel programming paradigm. There is no inter-process communication, and changing the size of the number of processors causes load imbalance. Reasoning about such things as buffer allocation is difficult or impossible [17,18]. In addition, the asynchronous nature of a program execution makes reasoning about synchronisation — and ultimately the concurrent behavior of the whole system — extremely complicated, if at all possible. What we need is fine-grained parallelism without the explicit use of locks, so that the model is safer to program in, and without asynchronicity, so that programs are easier to reason about. This, we believe, can be achieved through process-oriented design [19], built on the solid foundation of the process algebra CSP. Process Oriented Design uses synchronous channel communication, no locks, and no shared data. Each process is in full control of its own data; thus, no race conditions can ever occur. The following three points are important for optimal utilisation of multi-core processors; we believe they are provided by process-oriented design and programming with the system we describe in this paper.

- Architecture size independence: To fully utilise all the computing resources of a multi-core system, a concurrent program should consist of more processes than cores, at least if the code being implemented is not embarrassingly parallel.
- Verification: process-oriented design is based on the mathematical formalism of CSP and the π-calculus [20], which means that we can formally reason about the programs (the communication parts, at least) through the rigorous rules of the underlying mathematical model. We also can use tools, such as the model checker, FDR 2 [21], to prove properties about the code, including lack of deadlock or livelock.
- Efficiency: Extremely efficient runtime systems for multi-core architectures exist that support process-oriented programming [22,23], and we plan to make use of them.

These design principles and programming models were utilised on the Transputer, a CPU developed by INMOS, a British semiconductor firm, in the 1980s [24], these models were not mature for general computing at the time. The Transputer architectures were not like the multi-core architectures of today. As hardware matured, it became obvious that it is time to revive, re-invent and further develop the process-oriented programming/design model. In this paper, we describe the ProcessJ suite, including the design of a new language, tools for development (IDEs and graphical 'modular programming' interfaces), and built-in verification. In addition, we will develop teaching materials to make the model available to students, including a process component repository, online cloud storage, and a programming-in-a-browser tool. This combines online access to the compiler, and allows code to be run in the browser, with the possibility of incorporating it in a Web page with video clips as well as web-page text and pictures, something which is gaining momentum in the online teaching world.

1.3. Other CSP-Inspired Languages

occam is not the only language with channels and CSP-inspired semantics. One such language is XC [25], developed by Douglas Watt and David May (who, of course, was the designer of occam) at XMOS. As its name implies, XC has C-like syntax, with CSP-style extensions designed to take advantage of the XMOS processor architecture. XC programs are written for the XMOS architecture, and require XMOS emulators to run on other hardware platforms. Another CSP-inspired language is Go [26], developed by Robert Griesemer, Rob Pike, and Ken Thompson at Google. Go borrows the idea of channels from CSP, and has *goroutines* (coroutines) that communicate over these channels. Go channels are synchronous by default, but *are* buffered and can be specified as asynchronous, at which point the program becomes more Actor-inspired [27] than CSP-inspired.

2. Process-Oriented Design, CSP and Formal Verification

Building a programming model and language on top of a mathematical formalism like CSP provides an extremely robust foundation. We can actually prove things about the code we write, even without running it. Static proofs can be constructed for the absence of deadlocks or livelocks. A concurrent system can be verified before it is implemented, and different scenarios of behavior can be explored; best of all, no testing for concurrency errors is ever necessary. This alone should reduce the amount of time and money spent on testing; furthermore, we actually know more about the behavior of the code, after using the model checker, than we would by only testing it. Therefore, more confidence can be placed on the products developed.

The process-oriented programming model is a perfect fit for much of the technology being developed today, including robotics, UAVs, and other component-based systems. Many systems that would benefit from a multi-core CPU consist of semi-autonomous sub-systems that, if programmed using the standard thread-and-lock model, easily become an entangled mess that cannot be reasoned about. Unless Object Oriented programming models are significantly changed to behave exactly like the Actor model [27,28], with each object running in its own thread, there is not much to be gained [29,30,31].

In a process-oriented system such as the one we propose, sub-systems can be programmed independently, verified and tested; the composition of such sub-systems can be verified as well. Since CSP has composable semantics, so will the processes in the language. For example, there is no need for explicitly testing if an aircraft's control system fails to respond, and deadlocks, if the auto pilot is switched off at the same time as the flaps are extended. Such problems can be formally proven to be absent or present before the model is implemented.

To better appreciate the proposed project, it is worth focusing on what process-oriented programming and design is. The basic tenet is that a program consists of a number of processes, possibly executing concurrently; however, each executes in their own thread of control and each is 100% in control of its own state (data). Processes communicate through directed channels that are connected to a sender and a receiver – or multiple senders and receivers, in the case of shared channel ends. All communication is synchronous, something that enables us to reason and prove properties about our programs. Processes can be composed in sequence, e.g., statements in Java or C using the ";" operator; or in parallel, e.g., using the `PAR` construct in occam-π. We can model such a language — or at least the concurrency and the communication — by using the mathematical formalism CSP developed by Hoare [11,12], and extended by Roscoe [13]. FDR 2 [21], a formal model checker, can be used to statically verify properties about our programs.

Process composition facilitates *local reasoning*, that is, reasoning about a process at any level of composition. Local reasoning permits reasoning about a process, in isolation, without having to reason about the other processes it contains, or interacts with. Contrast this with Java's thread-based method invocation semantics, as described by Welch [29]. Threads of execution weave in and out of many objects as methods are invoked from inside the bodies of other methods. Objects are at the mercy of any other object that has a reference to it, and there is nothing to stop passing as a parameter the reference of one object to another (aliasing). Objects are passive; they are not in control of their own destiny. Methods may be invoked at any time, regardless of whether the object is ready. To reason about object behavior is to follow the spaghetti trails of threads of execution in and out of multiple objects; that is, one must reason about many objects to reason about the behavior of one object. This is not so with process-oriented design.

In addition to what we already described, CSP supports the idea of external choice. When executing a receive call on a channel, since the communication semantics is synchronous, a process is blocked until the sender is available to transfer data; however, this might not always be what a process wants. An external choice operator, often referred to as an alternation, is the ability to choose between several events — typically input, but also could be timeouts or barriers — and to nondeterministically choose from one of the events that are ready.

CSP reifies nondeterminism in a concurrent program by making external choice available explicitly through alternation. By avoiding alternation in a concurrent program, the program remains deterministic! In other words, process-oriented concurrent programs are deterministic by default! By giving the programmer the choice of when she needs to explicitly introduce alternation into a program, the programmer controls when a program may be nondeterministic!

With these seemingly simple primitives — parallel composition, synchronous communication, and alternation — we can write quite powerful concurrent code; even better, we can reason about it and prove properties, freedom from deadlock and livelock.

3. New Languages, Education and Teaching

One obstacle faced by new languages is that the general public does not use them. This poses a serious risk. However, we believe that since the idea of teaching concurrent computing is becoming more and more common, and since both the IEEE and the ACM are including parallel programming in their core curriculum [32], we believe that if a language is packaged with the right tools — integrated development environments, graphical programming tools, and online teaching tools — as well as a comprehensive library of support material (lesson plans, notes, slides, etc.), there is a much greater chance for a language to be used in a teaching setting. The more people who learn it, the greater the chance it will make its way into industry.

For this reason, to support teaching process-oriented design and programming in the classroom or to anyone who has an interest, we propose a project that integrates both a technical part — writing the software to support the new language and its tools — and an educational part that leverages an online approach. The project we propose includes developing a new process-oriented language, IDE/GUI tools, online programming tools, lecture plans, course descriptions, notes, and other support materials.

4. Proposed Work

The project we propose consists of two different yet intertwined parts: a technical and an educational part. In the technical part, an appropriate language and the aspects associated with implementing it will be developed. These aspects are not just about the compiler, but also include such topics as graphical programming environments, runtime systems, and automatic generation of scripts for code verification as well as an online process repository and cloud storage, enabling easy process reuse and sharing. A JavaScript runtime and a compiler back-end is proposed to interface with in-browser programming, an emerging trend in online education. A tool that could be helpful in implementing this is Emscripten [33], a source-to-source (LLVM to JavaScript) compiler, which translates LLVM bytecode, typically created from compiling C or C++, to JavaScript. This would be an obvious solution (at least as a proof of concept) as we plan to produce C++ code from the ProcessJ Compiler.

The educational part primarily will be concerned with developing materials supporting the teaching and learning of techniques related to process-oriented design and programming through the use of the ProcessJ language and the associated tools as well as the integrated Web Online Teaching Tool. Such materials can include lectures notes, slides, handouts, exercises, and supplementary materials. A book, we believe, will be a necessity, and videos would incorporate nicely into the online education experience.

Figure 1 illustrates the proposed ProcessJ suite. Boxes in light blue correspond to projects that will be undertaken as part of the proposed work, and boxes in gray are existing tools/resources that will be utilised in the project. Figure 1 contains three different types of arrows signifying what a sub-project produces, what it is based on, and which other sub-projects it uses; these three different types represent output, implemented interfaces, and inheritance.

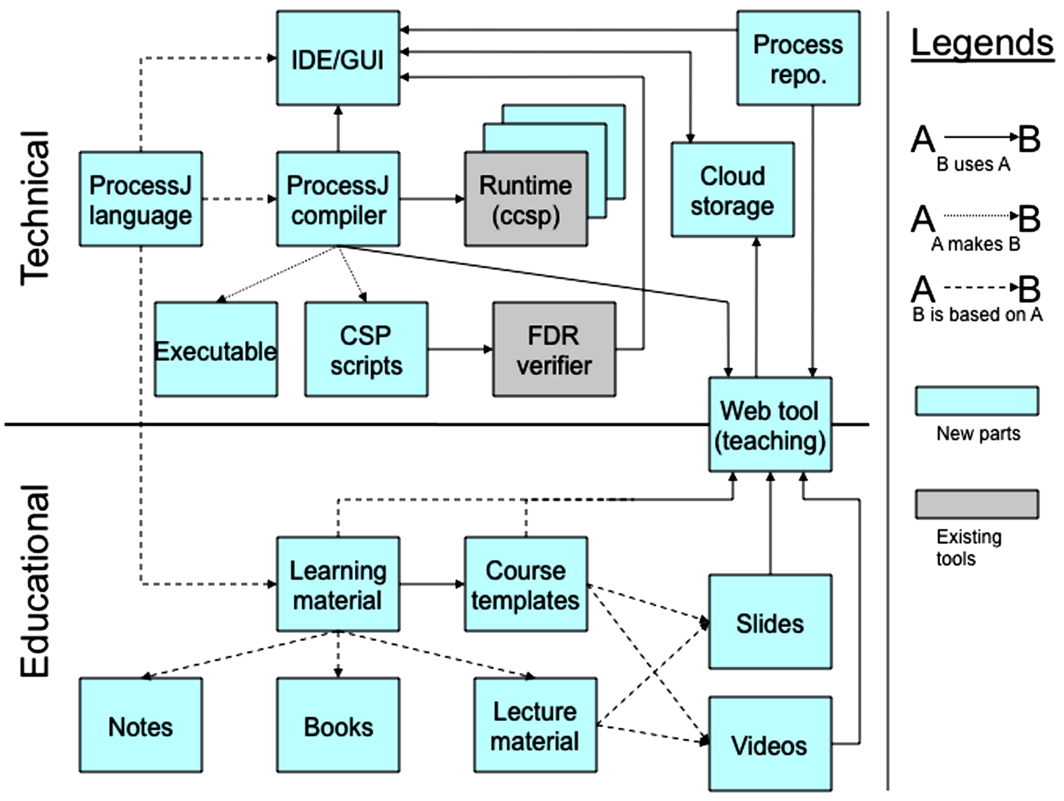


Figure 1. The overall proposed work of the project.

5. Technical Aspect

The technical part of this project is concerned with the actual software development, specification, and implementation. The following steps are to be undertaken:

- Language specification: Study existing popular languages (like C/C++ and Java) and develop a language specification that takes the familiar syntax and semantics of these languages into account for all similar syntactic constructs; that is, the syntax and semantics of a "while-statement" in ProcessJ is the same as in C/C++ or Java. New primitives required for concurrency — such as process, channels, barriers, and alternations — will have the same syntactic feel; however, the semantics will be that of CSP.
- Compiler construction: Implement the compiler for ProcessJ using modern compiler writing techniques and analysis. At first, we will utilise the CCSP/C++CSP [34,35,36] existing runtime systems to provide an efficient and reliable execution environment, later adding new runtimes or target architectures, such as clusters. One separate runtime in JavaScript is already included in the project.
- Graphical Programming Environment / Integrated Development Environment: Develop a graphical programming environment that supports the compositional and naturally layered approach that suits process-oriented design, including "programming by pictures" [37,38] and code reuse through a "LEGO catalog" method [39]. This IDE should interface with the compiler and the formal model checker, FDR 2, as well as an online repository for sharing ProcessJ code.
- Auto-generated Program Specification: The compiler should automatically generate CSP specifications of a program in machine readable form (CSP_M), enabling automatic verification of the absence or existence of deadlocks, livelocks, etc. [40]; furthermore, the support of specifying assertions about the program in order for the model checker to check should be supported [41].
- New back-end/runtime systems: Additional back-ends of the compiler targeting different execution architectures, such as clusters, using MPI, and the Java Virtual Machine, possibly using JCSP [19,42,43]. As mentioned we have already planned a runtime in JavaScript and a back-end code generator that produces JavaScript. Other interesting runtime systems include the LUNA CSP-Capable Execution Framework [44] and the light weight process MPI system [45] from the University of British Columbia.
- Online Repository: To encourage code reuse and code sharing, an online process source repository is planned, similar to the well-known app stores for mobile phones.
- Cloud Storage: To fully integrate the development and sharing of code in both the IDE and the online teaching tool (See next bullet point), a cloud storage can serve as a central hub for (private) code storage, making access available from any place with a network connection.
- ProcessJ Online Teaching Tool: A tool to tie the educational and the technical aspects of this project together is a Web based programming interface that can be placed on Web pages along with text, video, pictures, etc. This uses the JavaScript runtime, mentioned earlier, to facilitate execution of the compiled ProcessJ code directly in the user's browser. Such a tool will allow anyone to program in ProcessJ, and to execute code on any platform that contains a browser that supports JavaScript, albeit not as fast as on a multi-core architecture.

In the following subsections, we investigate each aspect in greater detail. In addition, we specify what experiences we have with each aspect topic, any work that has been done in the area, and what remains to be done.

5.1. Language Specification

There are many reasons why a language is not successful. Some languages are hard to learn and master, some have hard-to-understand semantic models, and others have poor syntax. Naturally, this assessment could be subjective, and subject to arguments. Others believe that languages who "made it" did so because they filled a void at just the right time in history.

We believe that now is the time for process-oriented programming to become popular; however, rather than make the mistakes of other languages developed throughout history, we are not going to make up a whole new set of syntactic rules, but rather take advantage of the knowledge that programmers already possess. Our aim is to produce a language specification that is syntactically similar to languages like C/C++ and Java, and which has the same semantics for all the well-known sequential language constructs, such as loops and variables. Then, we plan to introduce new process-oriented concurrency primitives — channels, communication, barriers, and mobile processes / channel-ends with their respective CSP/π-calculus semantics — while retaining a syntax that would be familiar, or at least reasonably obvious, to C/C++ and Java programmers.

CSP semantics apply to concurrency primitives like channel, channel-communication, barrier, and alternation (i.e., choice of which channel, out of many that may be possible, to engage). The semantics of the π-calculus apply to the mobility aspects of the language (mobile processes and mobile channel ends). Mobility is an area in which we are very interested and on which we have worked in the past. Our approach differs from the one (currently) taken by occam-π [4,5], for example, in the way we implement interfaces to the mobile processes. The occam-π approach uses a single interface for each mobile process, possibly requiring dummy actual-arguments for some parameters that might not be referenced; we propose a polymorphic approach, where each resumption of the mobile process uses a different interface, something we have (theoretically) investigated in [46,47] and intend to include in ProcessJ. At this point, we have a draft version of the ProcessJ language, which was used in [48]. Figure 2 shows a simple *producer-consumer* system written in ProcessJ, whose behaviour we shall return to in Section 5.4.

What remains to be done to complete this task includes developing a solid model for a package structure similar to Java's package model, finalising a reasonable syntax for structured literals, and determining how heap allocated values should be handled, for example whether to have explicit "free" or not, and if not, what sort of garbage collection the language should support. Although not part of the language directly, time must be spent on integrating into the ProcessJ language support for automatic verification; that is, an accompanying specification language must be developed. Some work already has been done on this incorporation with our research colleagues at the University of Kent at Canterbury [40,41].

5.1.1. Aliasing and Usage Checking

One of the holy grails (assuming we can have more than one) of occam is the non-aliasing semantics. This is something we strongly believe needs to be discussed, and perhaps rethought. One the one hand, no aliasing is an extremely nice feature: it prevents us from making an awful lot of mistakes, some potentially real bad. On the other hand, non-aliasing also prevents us from implementing a lot of very useful data structures (for example, circular lists).

We firmly believe that for a language to have a real impact, it is important that it is general enough to allow the programmer to actually write the data structures that are needed for the algorithms implemented. If non-aliasing within data structures prevents the language

```
proc void Producer (chan<int>.write out) {
    int x = 42;
    while (true) {
        while (x < 1000) {
            out.write (x);              // write to channel
            x++;
        }
        while (x > 0) {
            out.write (x);              // write to channel
            x--;
        }
    }
}

proc void Monitor (chan<int>.read in) {
    int last = in.read ();              // read channel
    while (true) {
        int x;
        x = in.read ();                 // read channel
        if (x == last) {
            ... system failure detected
        }
        last = x;
    }
}

proc void main() {
    chan<int> c;
    par {                               // in parallel do ...
        Producer (c.write);
        Monitor (c.read);
    }
}
```

Figure 2. A two process *producer-consumer* system in ProcessJ. The Monitor process checks that the `Producer` never sends the same number twice in succession (i.e., that `Producer` does not violate such a condition in its specification). A CSP model, verifying that this system failure will not happen, is shown in Figure 4.

from being fully utilised, then perhaps aliasing should be allowed in certain circumstances; but the programmer should be aware that, that in those circumstances, some things cannot be checked by the compiler. Not giving the programmer the freedom to design the data structures the application needs hampers the usability of the language, which is not an attraction.

We hope this topic will form the basis of a heated debate in the process oriented community — it is something that needs to be considered. It touches upon the age old issue with how strict compilers should be (how restrictive are the semantics of the language?) and how much responsibility can be safely given to the programmer. We strongly believe that a middle ground, with the possibility of support from the compiler when needed, is the best approach. We note that Barnes' design of Guppy [49] (a *spin-off* language from occam-π) and the discussion proposals from Welch [50] (in these proceedings) incline to this view.

As an important side note to aliasing is the issue of parallel usage checking; having the compiler guarantee no memory location is accessed concurrently in a way that might create a race hazard is a strong semantic check. It is an extremely useful feature to have, but unfortunately, it becomes even more difficult if aliasing is allowed. We have not yet decided on where to draw the line with respect to aliasing, but parallel usage checking is something that will impact this decision. However, part of the future research of this project is to establish a balance between semantic strictness of the language and the usability; both are extremely important subjects, and several papers on this subject alone could be written. Enough to say for now that we are aware of the issue and it will play an important role in defining the semantics of the language and the implementation of the compiler.

5.1.2. APIs

Arguably, one of the reasons occam has not enjoyed more widespread adoption (apart from its Transputer roots and indentation syntax, which was not fashionable until Python came along) is its lack of APIs for certain common utilities. For example, utilities that provide the ability to interact directly with the underlying OS and network, as well as access to common collections classes (arrays and records are not the only data structure programmers desire).

The CSP model, at its core, is composed of *sequential* processes. The ProcessJ API should permit access to processes written in native code (e.g., C for systems calls), as well as provide collections classes. We've previously discussed (in Section 5.1.1) the challenges and necessity of supporting aliasing. Perhaps wrapping such recursive data structures in a collections API is one way forward. Regardless, APIs must exist for a language to be useful, and they must be extensible and allowed to evolve with the feedback of the language's user community.

5.2. Compiler

A language without a compiler is, of course, a dead language; therefore, we propose the development of a new compiler written in C/C++ for speed, which, besides the regular parsing, scope, and type checker, etc., will produce code that can be linked with the CCSP/C++CSP runtime [35,36,22]; this is a highly efficient multi-core runtime system for process-oriented languages, developed and maintained at the University of Kent at Canterbury. We have authored papers [51,52,53,40,41,54,9] and done research with this group in the past, and we are actively working with them now. We anticipate a close collaboration between our two institutions and the Kent group. It should be noted that many of the people who have worked on several occam-π compilers — Fred Barnes and Peter Welch (KRoC [55]) as well as Neil Brown (CHP [56]) — are still associated with this group, something we view as an asset for support in the development process.

As a proof of concept, a Java version of the ProcessJ compiler was implemented by a UNLV M.Sc. student [48]; this implementation in Java was slow and extremely incomplete; it contained virtually no semantic analysis. As stated, a new compiler written in C++ is needed, and two research undergraduate students at UNLV currently are working on this project. We have a working scanner and parser as well as parse trees; however, no semantic analyses or code generation have been implemented. Part of the implementation of the compiler will include automatically producing CSP scripts; this will be described in Section 5.4.

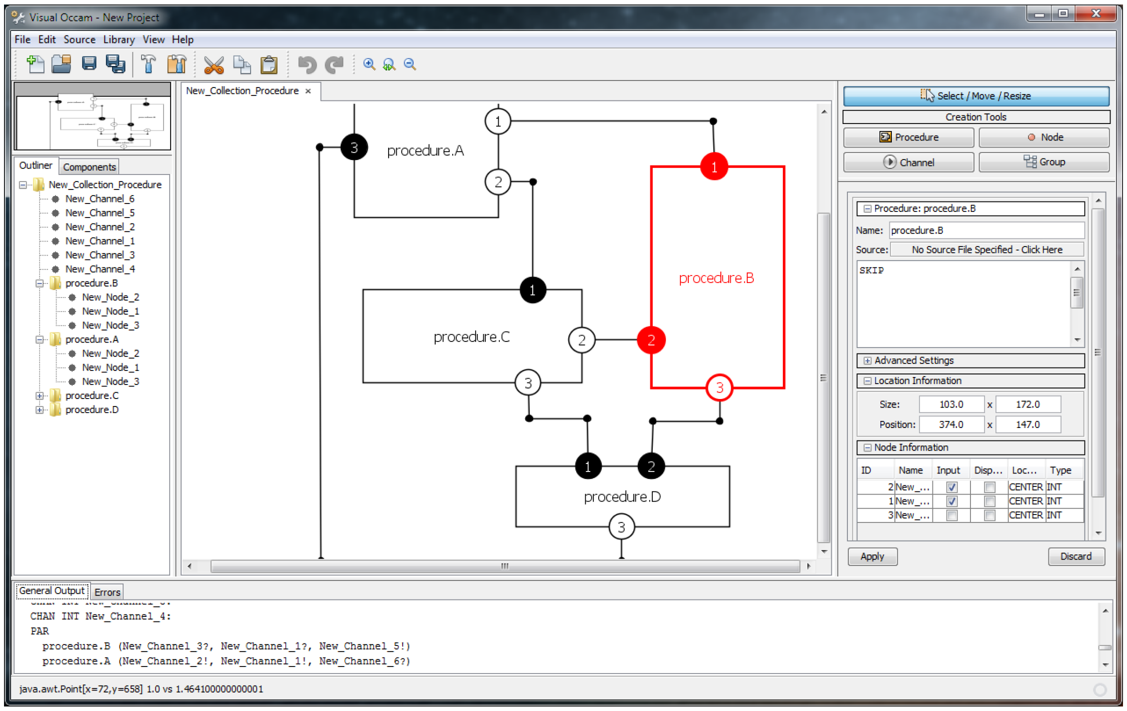
5.3. Graphical Programming Interface / Integrated Development Environment

Today, many popular programming languages, such as Java, C#, and Objective C, have very sophisticated programming environments: NetBeans for Java, Eclipse for Java and many other languages; Visual Studio for C#; and Xcode for Objective C. We believe that providing good programming support in terms of an integrated development environment can help boost a language's availability and reduce its learning curve. We have worked on a prototype for such an IDE for occam-π, called Visual occam [37] (see Figure 3). This prototype was by no means a finished product, but it taught us important lessons about what such an IDE must contain.

Since ProcessJ is a process-oriented language, the underlying model consists of processes that are composed in parallel, and which communicate through the use of channels.

This model is a perfect fit for a graphical programming tool, in which processes and channels as well as barriers are visualised. Figure 3 shows a number of processes and channels. A graphical tool like this can support a programmer in developing and maintaining a library of processes that can be reused; we often refer to this as the programmer's "LEGO catalog" because processes, like LEGO blocks, serve as building blocks for bigger structures: LEGO blocks click together, and processes compose by the means of channels and barriers. Once a programmer has a process, which itself can contain several levels of other concurrently composed processes, he or she can add such a process to the catalog of existing, reusable processes and, at a later date, simply drag it back into the IDE and reuse it.

In keeping with this idea, the online process source repository (Section 5.5) goes hand in hand with an IDE; it should integrate flawlessly with the "LEGO catalog" functionality of the IDE, allowing the programming to move already written processes from the repository to the local "LEGO catalog". Since verification is one of the pillars of process-oriented programming and development, the IDE should be able to interface with the compiler and also with the verification module (FDR 2). It should interpret the output of the verifier and relate it back to the source code, if possible. In other words, it should be possible to perform all the actions necessary from within the IDE; this ranges from process design at the source level, process layout (the boxes on the middle canvas in Figure 3), through to verification in the model checker (FDR 2). No work has been done yet specifically towards this sub-project. However, many important lessons were learned in the development of Visual occam, and we believe that we have a good idea of the pitfalls and what to do better in order to produce a highly usable IDE for ProcessJ. For a proof of concept that this kind of feeding back from the output of FDR to source level is possible, see the ASD:Suite by Verum [57]. FDR was integrated into their tool set and if a check for deadlock or livelock fails, the trace leading to it is presented in terms of a message sequence on the input model and its states and events.



Procedure: procedure.B

Name: procedure.B

Source: No Source File Specified - Click Here

```
SKIP
```

Size: 103.0 x 172.0

Position: 374.0 x 147.0

ID	Name	Input	Disp...	Loc...	Type
2	New_...	✓		CENTER	INT
1	New_...	✓		CENTER	INT
3	New_...			CENTER	INT

```
CHAN INT New_Channel_4:
PAR
  procedure.B (New_Channel_3?, New_Channel_1?, New_Channel_5!)
  procedure.A (New_Channel_2!, New_Channel_1!, New_Channel_6?)
```

```
java.awt.Point[x=72,y=658] 1.0 vs 1.464100000000001
```

Figure 3. Visual occam (M.Sc. thesis proof-of-concept programming GUI.)

5.3.1. Documentation

The graphical process diagrams can also be used as part of the documentation, perhaps along with a JavaDoc style of comments, to generate comprehensive (online) documentation; this is an area we have not considered extensible, but which will become fairly important for the usability and learning curve of ProcessJ and its tools.

5.4. CSP Specifications

As argued earlier, one of the strengths of a language based on a mathematical formalism like CSP is the ability to prove properties of the programs. ProcessJ will be based on CSP process algebra — and also the π-calculus for mobility (although the FDR tool does not currently support verification of mobility). Thus, a one-to-one correspondence between runnable code and verifiable specification will exist. This is further discussed in [40,41].

We believe that such a mapping can be automated and added to the ProcessJ compiler, like a new target language. Rather than targeting the CCSP/C++CSP runtime, we will produce CSP_M, which is the machine-readable form of CSP accepted by the model checker FDR 2. For example, Figure 4 shows the CSP model of the producer-consumer system in Figure 2, with an assertion that the system is deadlock-free added at the end.

```
Producer (out) = ProducerUp (out, 42)

ProducerUp (out, n) =
  if n < 1000 then
    c!n -> ProducerUp (out, n + 1)
  else
    ProducerDown (out, n)

ProducerDown (out, n) =
  if n > 0 then
    c!n -> ProducerDown (out, n - 1)
  else
    ProducerUp (out, n)

Monitor (in) = in?x -> MonitorLast (in, x)

MonitorLast (in, last) = in?x ->
  if x == last then
    STOP                      -- system failure detected
  else
    MonitorLast (in, x)

channel c : {0..1000}

Main = (Producer (c) [| {|c|} |] Monitor (c)) \ {|c|}

assert Main :[ deadlock free [F] ]
```

Figure 4. CSP_M specification for the *producer-consumer* system in Figure 2.

FDR verifies this assertion. This means that the `STOP` in `Monitor` can never happen, since otherwise there would be deadlock. In turn, this means that consecutive numbers output by `Producer` are never the same, since otherwise the `STOP` would have been reached. Therefore, in systems incorporating this `Producer` process, there is no need to program the run-time checks implemented by `Monitor` in any processes taking its production — we have *verified* that `Producer` fulfills this specification. So, production code can safely dispense with such run-time monitoring. This is a powerful win from formal analysis and deadlock checking.

CSP is powerful enough to prove properties such as *trace refinement* (i.e., that the event sequences the *refined* process can perform are event sequences of the original, usually called the *specification*) and *failure refinement* (i.e., that the sets of events the *refined* process can refuse, in any of its states, are sets of events also refused by the *specification*). Trace refinement establishes *safety*: only specified event sequences can happen. Failure refinement establishes *liveness*: if the specification says that certain events will happen (i.e., that they will *not* be refused), then they *will* happen in the refined process!

These are strong properties of the process algebra, which we think should be accessible from the ProcessJ language, so that verifictaion becomes part of normal programming practice. Such an approach should greatly narrow the chasm that currently exists between implementation and verification language. If the two are not significantly alike, proving that the formal model is a correct abstraction of the implemented code becomes almost as daunting as the verification process itself. We believe that by narrowing this gap and automating the process, there is a much greater chance that verification will be used more frequently.

As a side note, we have taught verification using FDR 2 and CSP_M in courses. Students find these tools which we use to verify programs almost as easy to understand as the programs themselves. The ultimate goal is to reach a state where verification is just programming, and we think we can do that.

No work pertaining to the proposed project has been undertaken to date; however, early investigation has begun. Currently we are working on a research paper with the Kent group concerning this subject that should lay a solid foundation for completing this sub-project.

5.5. The ProcessJ Process Repository

As described in Section 5.3, in addition to a local LEGO catalog within the IDE, we propose an online repository—much like the Apple App-store and Google's Android Market, where developers can share source code for processes they have written. Such a repository should integrate into the IDE as well as the online tool set, and also have a searchable/browsable Web interface. Users can upload their ProcessJ code for other programmers to use. Processes are divided into categories so they can be browsed either through a Web interface (directly or through the ProcessJ Online Teaching Tool, allowing for transfer from the repository to the user's own private cloud storage—See Section 5.8) or through the IDE. Such a repository should of course be searchable. Furthermore, we propose the possibility of associating CSP_M verification certificates with uploaded code. That is, when a user uploads a piece of source, each process can have CSP code associated with it, and in addition typical things like deadlock, livelock etc. can be marked as 'checked'. This enables other users to know about certain properties of the code before they download and use it.

The process repository does not require any development of new tools, but simply an implementation using HTML and also a database. However, we believe it forms a key feature in making ProcessJ a common language: the more libraries and publicly available code, the easier the development gets of new code. No work has been done on this either; however, this can be undertaken as a completely independent project from much of the other development that is required for the project.

5.6. Multiple Back-Ends/Architectures

As proposed in Section 5.2, the first iteration of the ProcessJ compiler will target the CCSP/C++CSP runtime; that is, it will generate C or C++ code that gets compiled and linked with the runtime using the gcc or g++ compiler. The CCSP/C++CSP runtime is specifically developed and optimised for multi-core architectures; however, as already seen, we might need to generate code for different targets/back-ends. We have already mentioned two: CSP_M for use in the FDR 2 model checker and JavaScript for the ProcessJ Online Teaching Tool. To make process orientation more available to other platforms/targets/architectures, we have considered a number of possible other back-ends:

- Instead of generating C/C++ and linking with the CCSP/C++CSP runtime, we could generate Java and link with the JCSP [43] library. This would make process-oriented programming available on architectures that do not support the CCSP/C++CSP runtime; not particularly efficient, because it is built upon Java threads.
- Clusters that often use MPI [58] (and C) could also run process-oriented code. The compiler would then generate C with MPI calls rather than CCSP/C++CSP calls. A particularly interesting possibility is the MPI implementation from The University of British Columbia [45] which supports much finer grained levels of parallelism than the regular MPI implementations.
- Other virtual machines could be targeted, such as the LLVM [59]. In [22], a translator for ETC (Extended Transputer Code) to LLVM was developed.

Countless other targets exist, but we will focus on the MPI and the JCSP targets.

5.7. The ProcessJ Online Teaching Tool (Technical Part)

The recent success of massive open online courses (MOOCs) in attracting student interest from around the world is impossible to ignore. With the use of recent technologies incorporated into major Web browsers under the umbrella term of HTML5, it is now possible to offer highly interactive online computer science courses, which run entirely within a student's Web browser.

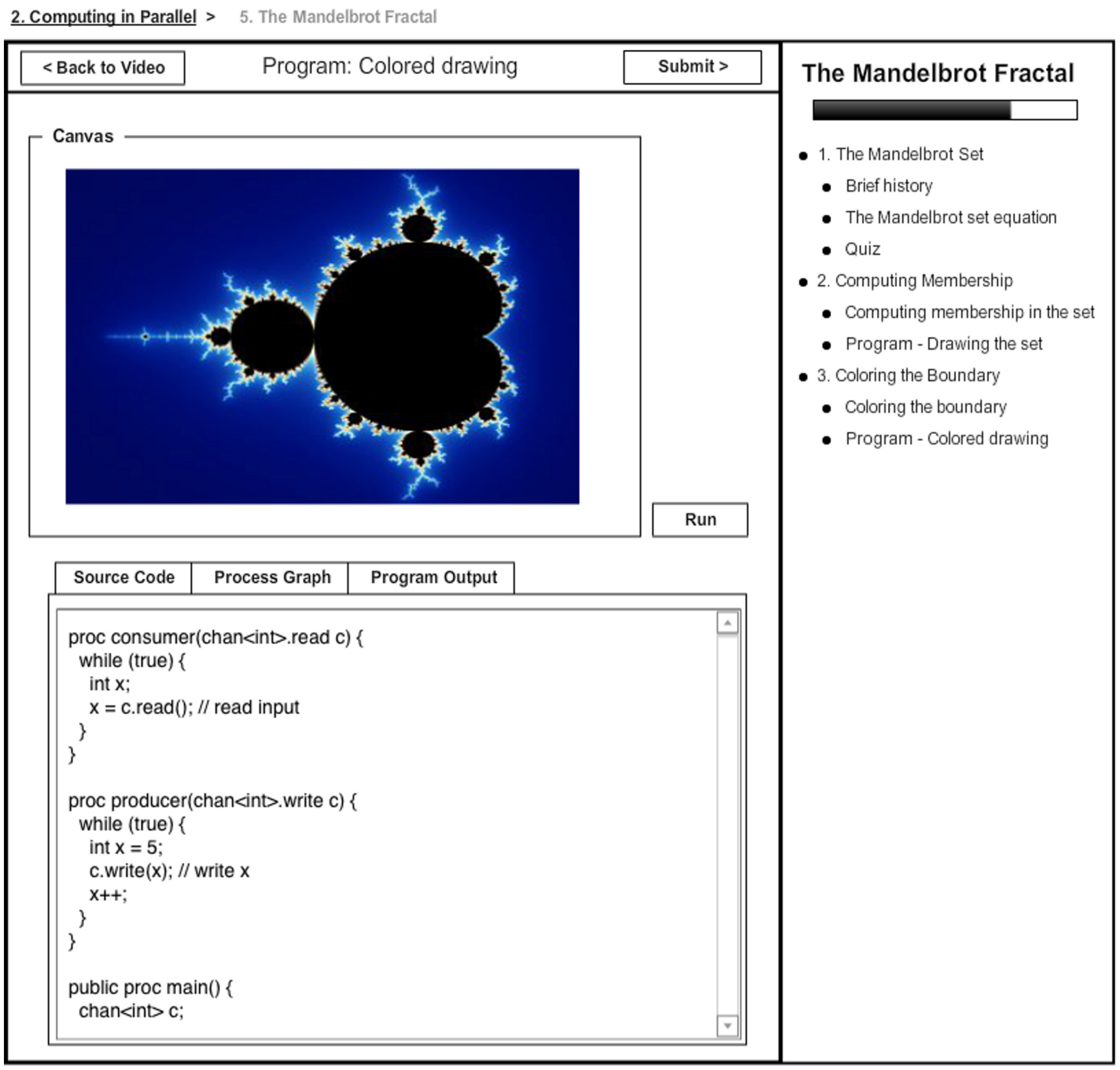
By producing a ProcessJ compiler back-end and runtime system that targets JavaScript, ProcessJ code can be executed directly within the Web browser environment. The core of the ProcessJ Online Teaching Tool will consist of four main components:

- The code editor/debugger interface for writing ProcessJ programs in the browser.
- A ProcessJ to JavaScript compiler back-end, which will run on a server and compile programs on demand.
- The ProcessJ JavaScript runtime system, which will provide concurrent execution of processes in the single-threaded JavaScript interpreter using co-operative multitasking.
- A ProcessJ API for the HTML5 canvas for interactive graphical programs.

In addition to the online programming environment, the ProcessJ Online Teaching Tool will provide access to learning materials and provide a framework for automated grading of students' programming exercises. With these features in combination, self-contained online classes can be produced that can be accessed by students in any place and at any time. Figure 5 shows a mock-up of the ProcessJ Online Teaching Tool. We discuss similar browser-based IDEs and online learning environments (for other languages) in Section 6.3.

5.8. Online Cloud Storage

Since an integral part of the ProcessJ suite is the online programming environment, which enables a programmer to develop ProcessJ code from any Web browser, the issue of code storage arises. To make available a user's code in any Web browser (on any computer with a Web browser that supports JavaScript and is connected to the Internet), code cannot be stored locally (at least not as the only location for the code).



The Mandelbrot Fractal

- 1. The Mandelbrot Set
  - Brief history
  - The Mandelbrot set equation
  - Quiz
- 2. Computing Membership
  - Computing membership in the set
  - Program - Drawing the set
- 3. Coloring the Boundary
  - Coloring the boundary
  - Program - Colored drawing

```
proc consumer(chan<int>.read c) {
  while (true) {
    int x;
    x = c.read(); // read input
  }
}

proc producer(chan<int>.write c) {
  while (true) {
    int x = 5;
    c.write(x); // write x
    x++;
  }
}

public proc main() {
  chan<int> c;
```

Figure 5. A Web mock-up of the ProcessJ Online Teaching Tool (PJOTT).

We propose an online cloud storage (in the style of Dropbox [60], though not user managed to the same extent), that integrates the online programming environment, and the integrated development environment as well as the Online Repository.

A user logs into his or her cloud storage and the code is immediately available to the in-browser development environment; similarly, logging in within the IDE will make all the cloud storage code available.

Furthermore, the cloud storage makes getting code from and putting code into the Online Process Library an easy task. Cloud storage enables additional possibilities, and we envision many possible extensions. For example versioning and sharing. As the project develops, more extensions will surely be included.

One possible implantation of this could be on top of Google Drive, using the Google Drive SDK [61]; alternatively, GitHub [62] could be utilised through the graphical user interface, or even the ProcessJ website. There are many ways to implement online cloud storage, and implantation wise, it might be a great idea to allow a plug and play situation, where the user can decide which system he wants to use (Google Drive, GitHub, Dropbox etc.)

6. Educational Aspect

Every ten years, an ACM-IEEE Computer Society joint task force puts out an updated document containing guidelines for Computer Science curricula; the latest version, Computer Science Curricula 2013 (CS2013) [32], is due later this year. The latest draft (Ironman Draft Version 0.8) includes information about important (and exciting!) changes for the curriculum, especially regarding what topics are considered "core". The last time full guidelines were published in 2001 [63], and in interim updates [64,65], parallel computing wasn't identified as its own Knowledge Area within the Body of Knowledge. The story is different this time around: in 2012, the joint task force stated, when questioned about new topical areas that should be added to the Body of Knowledge, survey respondents indicated a strong need to add the topics of Security as well as Parallel and Distributed Computing . . . CS2013 includes these two new KAs (among others): Information Assurance and Security, and Parallel and Distributed Computing [32]. Concurrency should not be a last resort to gain performance, but should be taught as a fundamental model of interaction between processes. Supporting such a way of thinking both from an educational side as well as a programming-language side, for example, the use of a process-oriented language, makes concurrency as easy to approach for students just beginning their programming careers as any other sequential construct; one might argue that fully understanding state and mutation is even harder than synchronously communicating processes.

It is not a surprise that parallel computing has been reified in CS2013; Moore's Law has plateaued over the last decade, replaced by a focus on Amdahl's Law [66] to promote the development of ever-larger multi-core processors. As discussed in [67,68], the free lunch is over. In other words, in order to continue enjoying increased performance, we have to learn to harness the power of multi-core processors. We can no longer count on a faster processor coming out next year. Our challenge is to write better parallel programs and also to teach the next generation of programmers and computer scientists how this can be done.

Assuming we are completely successful in developing ProcessJ and its supporting tools, that wouldn't be enough to call it a day. We must provide a compelling set of materials for courses and course modules for faculty to use in the classroom. We must provide a path for teachers and professors to learn this material well enough that they can teach it to their students. To help get the word out, once the ProcessJ compiler is ready for broader use, we plan to offer workshops at parallel computing conferences, including the formerly named *Supercomputing (SC)* conference, now named the *International Conference for High Performance Computing, Networking, Storage and Analysis*; and the *International Parallel and Distributed Processing Symposium (IPDPS)*. Once the ProcessJ Online Teaching Tool is ready, we plan to offer workshops at such computer science education venues as SIGCSE and one or more of the regional CCSC conferences. We have some experience, through the CPA community, in presenting during teaching tracks and tutorials at conferences.

6.1. Learning Materials

Even as parallel processing enjoys a higher profile and recognition as a new Knowledge Area and core topic within the CS2013 guidelines, much of what people know about the topic is either limited — parallel and distributed computing is not everyone's area of expertise — or misunderstood — what is currently popular or being promoted regarding multi-core programming is not the only approach to solving these problems. Some people may think that threads and locks as well as shared memory is the only approach to multi-core programming. Others may know about asynchronous message passing, but not synchronous message passing.

The CSP model and process-oriented design, despite its long history, is still not in the mainstream, and what people think they know about it is wrong. It is our responsibility to provide learning material that both educates and re-educates our prospective audience regarding Process Oriented Design, while accommodating the different learning styles and environments of today's faculty and students. We will provide sample lecture notes, assignments, and video tutorials that present individual topics as well as demonstrate the use of the ProcessJ compiler and online teaching tool. These materials will be accessible online through a Web browser accessing the ProcessJ Web pages. We also will provide an online community space for teachers and practitioners to ask questions and exchange ideas, following the model of BlueJ's Blueroom [69].

The biggest obstacle we must overcome in our learning materials is the long-standing reputation that concurrent programming is hard, error prone, difficult to get right, and virtually impossible to reason about. For example, Muller and Walrath, in an article by Sun Microsystems on Threads and Swing [70] state:

> *"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."*

This may, in fact, be a well-deserved reputation in the mainstream of concurrent processing, but it doesn't apply to process-oriented design. The challenges of unwanted nondeterminism, race hazards, and deadlock do not go away; they simply don't enter the picture in the first place. Our materials must illustrate that process orientation is the right model and abstraction for modeling our parallel world.

We have some very elegant, time-tested approaches to teaching process-oriented design, using Welch's 'LEGO' catalog method approach to processes [39]. We have used such libraries of processes in our courses for the last decade of teaching our own concurrent programming courses, and will implement this library for the new ProcessJ language. These programs and exercises build parallel programming skills on top of students' existing sequential programming skills so that parallel composition becomes just another programming abstraction, like sequential composition, conditional statements, recursion and iteration. Because processes compose, process-orientation is a natural way to deal with complexity, including underlying parallelism. This approach also illustrates how we write parallel programs at much finer-grained level of parallelism than prevailing conventional wisdom currently believes is feasible.

6.2. Course and Module Templates

The CS2013 guidelines are broken down into Knowledge Areas that do not necessarily correspond to full courses. This allows for maximum flexibility for departments to design their curricula according to their particular levels of staffing, expertise, and numbers of students. As mentioned earlier, Parallel and Distributed Computing is a new Knowledge Area, but not all programs will decide to add a dedicated course in parallel programming. One option is to distribute parallel programming topics among several existing courses, for example, Algorithms, Operating Systems, Computer Architecture, and (we hope) introductory courses. This means we need to provide building blocks that can be put together into a single course or spread out among courses in existing curricula; the more self-contained we can make these building blocks, the more flexibility faculty and departments have in using the materials we provide. We will offer modules that can be used in a variety of courses, including algorithms, architecture courses, operating systems courses, and software engineering. We will use the CS2013 guidelines as a source for categorising different modules; in this way, faculty and departments can most easily use our ProcessJ materials in conjunction with the CS2013 document to revise their curricula to accommodate increased coverage of parallel processing material. We also will recommend different possibilities for a dedicated course in parallel processing, stressing the value of process-oriented design.

6.3. The ProcessJ Online Teaching Tool (Educational Part)

With the ProcessJ Online Teaching Tool (PJOTT), the technical and educational parts of this project intersect. PJOTT will provide students with a browser-hosted interface to access ProcessJ and its planned online resources. It also will make using ProcessJ in the classroom more convenient for faculty with limited resources because it won't be necessary to arrange for the installation and configuration of another programming language or IDE in their labs. Browser accessibility means faculty and students can focus on process-oriented design from Day One. This intersection of the technical and educational provides an important feedback loop for improving the tool and its interface for students, based on ongoing feedback. Another advantage of providing a browser-based IDE is that changes to PJOTT are centralised on the server, and don't require pushing out changes for the users. Each time the browser loads PJOTT, users access the latest version. This is good, because it takes time, effort, and user feedback to develop a well-engineered interface that follows pedagogically sound guidelines for online learning tools. What good is a lot of information if it is not organised and presented in an intuitive manner?

Other languages now have browser-based online learning playgrounds. Some examples include GoLang [71], Codecademy [72], and CodingBat [73]. While GoLang specialises in the Go language, Codecademy and CodingBat support learning multiple languages. These online learning environments have some similarities but different missions and goals. CodingBat provides a set of problems to be solved, and an editor with a run button for users to try to solve them. The teaching takes place elsewhere; CodingBat is a place to practice one's Java or Python coding skills. Codecademy provides courses for a larger set of languages using tutorial style lessons. These tutorials including an editor for users to type in their solutions each step along the way, and check for correctness. Codecademy attracts both students and teachers, providing a framework for both. Codecademy also provides a playground for several languages for users that just wish to have access to these languages from their browser. GoLang is a portal to all things Go-related, including providing an online editor and interpreter, and examples of programs that can be run from the browser. These existing online learning resources provide proofs of concept for some of what we plan to do for the PJOTT; however, we envision a deeper and more comprehensive integration with the teaching materials.

7. Conclusions and the Environment

We would like to conclude our paper, but we also feel it would be helpful to comment on the environment of this proposed work. Hoare introduced through CSP the notion of reasoning about the meaning of a computation through reasoning about its trace of observable events recorded by an Olympian observer. These observable events were offered by the *Environment*, and engaged in by the processes composed in a process network. Process-oriented design as a programming paradigm also exists within a larger environment, and if it is to succeed and become widely adopted, the process-oriented community must find a compelling way to communicate its ideas within this environment: the greater concurrency and high performance computing community.

7.1. Conclusions

We have presented what we believe to be a complete system for a new process-oriented programming system that is extremely well suited to take full advantage of today's multi-core architectures. This system includes a new process-oriented language, ProcessJ, rooted in the mathematical process algebra CSP. Programs written in ProcessJ are verifiable with respect to their concurrency. In addition, a programming environment and online code libraries and online storage are proposed as well as an online teaching tool, which allows for in-browser programming.

From an educational point of view, we propose creating course descriptions, support material, notes, and slides; in other words, anything needed to teach a course about process-oriented programming and design at the college level. The online teaching tool utilises these techniques to make such course material available to anyone with a Web browser. We believe this can be a very exciting project; the modularity of it means that a great many people and institutions can be involved with the development at the same time. We strongly believe that if the entire community pulls in the same direction, towards the completion of a suite like described in this paper, we stand a much better chance of 'reviving' process-oriented programming for the masses.

7.2. The Environment

This paper also formed the basis for an NSF proposal for around $750,000; however, the proposal was not funded, but three reviews were returned, and since this is a paper about a possible future of process-oriented programming, we would not do it justice without reporting the major issues found by the reviewers. We believe this can not only help shed some light on how to approach this project, but also give some ideas about what the scientific community (who are interested in concurrent programming) in general thinks about process-oriented design.

It seems that the scientific computing community thinks only in terms of HPC and shared memory, and if they ever deviate from that, MPI is as far as they are willing to venture. One of the comments was:

"At times, the proposal is talking about CSP as a panacea. Unfortunately, this (in my opinion) reveals ignorance of the richness of both the problem space (types of applications) and the solution space (known models for parallel or concurrent computation), and thus undermines the credibility of success. For example, try writing a well-performing parallel matrix multiplication in CSP."

We believe that such statements actually shows the lack of understanding by the reviewer (and thus we might conclude by the broader community as a whole) both as to what CSP is (it's not a programming language!) and that *noone* is proposing a panacea. We are not suggesting that all problems be solved with process-oriented design, but that the tasks for which process oriented design is tailored may be solved by using it — and that the breadth of application be explored. Apart from the fact that a version of matrix multiplication could be implemented extremely efficiently using a systolic array (which could be programmed easily using process-oriented programming techniques), we are not proposing that process-oriented design and programming should be utilised for all scientific and HPC programming; but we are advocating that it be used for embedded system like *setups*, where either the hardware directly maps to processes or, on shared memory systems, where the problem naturally decomposes to processes.

We believe there is a perception that after deriding parallel computing as not necessary to perpetuate the prophesy of Moore's Law for 30+ years, now that we have hit the wall for clock speed and multicore has come along, they are going to deride concurrency and make it a scapegoat if it now cannot perform that service. But we know there are limitations to speed-up, which gives rise to the speed-up myth (solve a problem of size N on one processor in time N means we should be able to solve a problem of size N on P processors in time N/P). Some problems are more amenable to speed-up than others. They look at matrix multiplication as an embarrassingly parallel (or more politically correct these days, "pleasingly parallel") algorithm; but that is based on a shared memory model of concurrency. Process-oriented design and programming is not a shared memory model, although it works extremely well on shared-memory platforms [22,23]! If your problem is matrix multiplication, and you have the cores, you may use a shared memory model — but communicating processes may also work (and be simpler). However, most problems do not have embarrassingly parallel solutions and their needs are for interesting and rich patterns of interaction between the sub-components that will solve them. Process orentation is looking to address this more general class of problem, as well as those more regular systems whose needs for memory vastly exceeds that which can be supported by a single memory bus fought over by a host of processor cores.

Programming-with-pictures is an extremely important approach to process-oriented design, and its strengths must be emphasised in the future. A lack of understanding of the importance of composability is shown by comments such as:

> *"The inclusion of a graphical interface / IDE is a nice idea, but orthogonal as a research problem."*

and:

> *"Graphical IDEs have been around forever, but have never made it beyond the fringe. They tend to survive only if supported by a monopolistic proprietary owner."*

Finally, it seems that the community does understand the importance of verification, and they do seem familiar with the strengths of CSP, but they often fall short with comments like:

> *"Can't you just use Google's Go?"*

In conclusion to these reviews and comments, we believe that the following issues must be emphasised in the future to better promote process-oriented programming and design:

- Process-oriented programming and design is *not* trying to be a panacea; we do not propose implementing all algorithms in this fashion, but we do propose using the design and programming techniques to those problems that naturally fit the paradigm; these include, but are not limited to, embedded systems and safety critical concurrent systems, such as control systems for machinery, aircraft, etc.
- Reusability and composability are important; and communicating processes, in particular, can be separately reasoned about, verified and reused.
- The relationship between verifiability, composability and code reuse does not seem to be recognised and must be emphasised.

Acknowledgments

The authors wish to thank the editors and the many helpful comments and references provided by our reviewers. Their careful comments and valuable feedback helped improve the content and presentation of this paper.

References

[1] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[2] Dick Pountain and Michael D. May. *A Tutorial Introduction to occam Programming*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[3] Michael D. May. CSP, occam and Transputers. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 75–84. Springer Verlag, April 2005.

[4] Peter H. Welch and Frederick R. M. Barnes. Communicating Mobile Processes: introducing occam-π. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[5] Frederick R. M. Barnes and Peter H. Welch. Communicating Mobile Processes. In Ian East, Jeremy Martin, Peter H. Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures 2004*, volume 62, WoTUG-27 of *Concurrent Systems Engineering Series, ISSN 1383-7575*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[6] Peter H. Welch and Frederick R. M. Barnes. Mobile Barriers for occam-π: Semantics, Implementation and Application. In Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, volume 63, WoTUG-28 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[7] Peter H. Welch and Frederick R. M. Barnes. A CSP Model for Mobile Channels. In *Communicating Process Architectures 2008*, volume 66, WoTUG-31 of *Concurrent Systems Engineering Series*, pages 17–33, Amsterdam, The Netherlands, September 2008. IOS Press. ISBN: 978-1-58603-907-3.

[8] Peter H. Welch. *An occam-π Quick Reference Guide*. Programming Languages and Systems Research Group, University of Kent, `https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference/`, 2011.

[9] Peter H. Welch and Jan B. Pedersen. Santa Claus: Formal Analysis of a Process-oriented Solution. *ACM Trans. Program. Lang. Syst.*, 32(4):37, April 2010.

[10] Peter H. Welch, Kurt Wallnau, Adam T. Sampson, and Mark Klein. To Boldly Go: an occam-π Mission to Engineer Emergence. *Natural Computing*, 11(3):449–474, September 2012. `http://dx.doi.org/10.1007/s11047-012-9304-2`.

[11] Charles A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[12] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[13] Andrew W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[14] Andrew W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[15] John A. Trono. A New Exercise in Concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994.

[16] Mordechai Ben-Ari. How to solve the Santa Claus Problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.

[17] Alex Brodsky, Jan B. Pedersen, and Alan Wagner. On the Complexity of Buffer Allocation in Message Passing Systems. *Journal of Parallel and Distributed Computing*, 65(5):692–713, June 2005.

[18] Jan B. Pedersen, Alex Brodsky, and Jeffrey Sampson. Approximating the Buffer Allocation Problem Using Epochs. *Journal of Parallel and Distributed Computing*, 68(9):1263–1282, September 2008.

[19] Peter H. Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000. ISBN: 1-892512-52-1.

[20] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[21] Formal Systems (Europe). *FDR2 (Failures, Divergence, Refinement) model checker*. `http://www.fsel.com/software.html`.

[22] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009.

[23] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. *Science of Computer Programming*, 77(6):727–740, 2012. Revised from LNCS (5521) paper with same name.

[24] Hamid R. Arabnia. The Transputer Family of Products. In A. Kent and J.G. Williams, editors, *Encyclopedia of Computer Science and Technology*, 1998.

[25] XMOS. XC Programming Guide. `http://www.xmos.com/published/programming-xc-xmos-devices`. [Online; accessed 01-August-2013].

[26] Google. The Go Programming Language. `http://golang.org/doc/`. [Online; accessed 01-August-2013].

[27] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. Actor induction and meta-evaluation. In *In ACM Symposium on Principles of Programming Languages*, pages 153–168, 1973.

[28] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323364, June 1977.

[29] Peter H. Welch. Java Threads in the Light of occam/CSP. In Peter H. Welch and André W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 259–284, Amsterdam, The Netherlands,

April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[30] Peter H. Welch and Jeremy M. R. Martin. A CSP Model for Java Multithreading. In *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*. IEEE Computer Society, jun 2000.

[31] Peter H. Welch and Jeremy M. R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and André W. P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 66, WoTUG-31, pages 275–301, Amsterdam, The Netherlands, sep 2000. IOS Press. ISBN: 978-1-58603-907-3.

[32] ACM/IEEE-CS Joint Task Force for Computing Curricula. Computer Science Curricula 2013, Ironman Draft (Version 0.8). Technical report, November 2012. `http://ai.stanford.edu/users/sahami/CS2013/`, [Online; accessed 01-August-2013].

[33] Alon Zakai. Emscripten. Technical report. `http://emscripten.org`, [Online; accessed 01-August-2013].

[34] Neil C.C. Brown. C++CSP2: a Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65, WoTUG-30 of *Concurrent Systems Engineering Series*, pages 183–205, Amsterdam, The Netherlands, "July" 2007. IOS Press.

[35] Neil C. C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 139–156, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[36] James Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[37] Mikolaj Slomka. Visual Occam: High Level Visualization and Design of Process Networks. Master's thesis, Department of Computer Science, University of Nevada Las Vegas, 2010.

[38] Maarten M. Bezemer, Robert J. W. Wilterdink, and Jan F. Broenink. Design and Use of CSP Meta-Model for Embedded Control Software Development. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan B. Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 185–199. Open Channel Publishing, August 2012. ISBN 978-0-9565409-5-9.

[39] Peter H. Welch. Concurrency Design and Practice. `http://www.cs.kent.ac.uk/projects/ofa/sei-cmu/`, November 2007. A course on concurrency and process-oriented design, based on occam-π.

[40] Peter H. Welch, Jan B. Pedersen, Frederick R.M. Barnes, and Carl G. Ritson. Self-Verifying Concurrent Programming. Presentation to IFIP Working Group 2.4, September 2011. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/IFIP_WG24`.

[41] Peter H. Welch, Jan B. Pedersen, Frederick R.M. Barnes, Carl G. Ritson, and Neil C.C. Brown. Adding Formal Verification to occam-π. In *Communicating Process Architectures 2011*, volume 68 of *Concurrent Systems Engineering Series*, page 379, Amsterdam, The Netherlands, June 2011. IOS Press. *Endnote address.* ISBN 978-1-60750-773-4.

[42] Peter H. Welch, Neil C.C. Brown, James Moores, Kevin Chalmers, and Bernard Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN: 978-1-58603-767-3.

[43] Peter H. Welch and Paul D. Austin. *Communicating Sequential Processes for Java (JCSP) Home Page*. Systems Research Group, University of Kent, 2010.

[44] Maarten M. Bezemer, Robert J. W. Wilterdink, and Jan F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In Peter H. Welch, Adam T. Sampson, Jan B. Pedersen, Jon M. Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam, November 2011. IOS Press BV. ISBN 978-1-60750-773-4.

[45] Sarwar Alam, Humaira Kamal, and Alan Wagner. Service-Oriented Programming in MPI. In Peter H. Welch and Jan B. Pedersen and Kevin Chalmers and Adam T. Sampson and Frederick R. M. Barnes, editor, *Communicating Process Architectures 2013*. Open Channel Publishing, August 2013.

[46] Jan B. Pedersen and Matthew Sowders. Static Scoping and Name Resolution for Mobile Processes with Polymorphic Interfaces. In *The thirty-third Communicating Process Architectures Conference, CPA 2011, organised under the auspices of WoTUG, Limerick, Ireland, June 19-22 2011*, pages 71–85, 2011.

[47] Matthew Sowders and Jan B. Pedersen. Mobile Process Resumption In Java Without Bytecode Rewriting. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, July 2011.

[48] Matthew Sowders. ProcessJ: A Process Oriented Programming Language. Master's thesis, Department of Computer Science, University of Nevada Las Vegas, 2011.

[49] Frederick R. M. Barnes. Towards a New Language for Concurrent Programming. *CPA 2011 Fringe:* slides at `http://www.wotug.org/papers/CPA-2011/Barnes11/Barnes11-slides.pdf`, August 2011. Guppy Home Page: `http://frmb.org/guppy.html`.

[50] Peter H. Welch. Life of occam-π. In *Communicating Process Architectures 2013*. Open Channel Publishing, August 2013.

[51] Neil C. C. Brown and Marc L. Smith. Representation and Implementation of CSP and VCR Traces. In *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 329–345, Amsterdam, The Netherlands, September 2008. WoTUG, IOS Press.

[52] Neil C. C. Brown and Marc L. Smith. Relating and Visualising CSP, VCR and Structural Traces. In Peter H. Welch, Herman W. Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner G. Styles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, volume 67 of *Concurrent Systems Engineering*, pages 89–103, Amsterdam, The Netherlands, November 2009. WoTUG, IOS Press.

[53] Jan B. Pedersen and Peter H. Welch. Concurrency, Intuition and Formal Verification: Yes, We Can! In *Workshop on Curricula for Concurrency and Parallelism, SPLASH 2010*, October 2010. Position paper and slides: `ftp://ftp.cs.ukc.ac.uk/pub/phw/sei-cmu/occam/papers/README.txt` [Online; accessed 01-August-2013].

[54] Peter H. Welch and Jan B. Pedersen. Santa Claus - with Mobile Reindeer and Elves. In *Fringe Presentation at Communicating Process Architectures conference*, September 2008.

[55] Frederick R. M. Barnes, Peter H. Welch, Jim Moores, and David C. Wood. *The KRoC Home Page*. Programming Languages and Systems Research Group, University of Kent, `http://www.cs.kent.ac.uk/projects/ofa/kroc/`, 2010.

[56] Neil C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H. Welch, Susan Stepney, Fiona A. C. Polack, Frederick R. M. Barnes, Alistair A. McEwan, Gardner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 67–83, Amsterdam, The Netherlands, September 2008. WoTUG, IOS Press.

[57] Verum Software Technologies B.V. The ASD:Suite. `http://www.verum.com/`, 2013. [Online; accessed 01-August-2013].

[58] Jack Dongarra. MPI: A Message Passing Interface Standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.

[59] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[60] Dropbox. Online Cloud Storage. `http://www.dropbox.com/`. [Online; accessed 01-August-2013].

[61] Google. Google Drive SDK – Google Developers. `https://developers.google.com/drive/`. [Online; accessed 01-August-2013].

[62] GitHub - Build Better Software Together. Technical report. `http://github.com`, [Online; accessed 01-August-2013].

[63] ACM/IEEE-CS Joint Task Force on Computing Curricula. ACM/IEEE Computing Curricula 2001 Final Report. Technical report, 2001. `http://www.acm.org/sigcse/cc2001` [online; accessed 01-August-2013].

[64] ACM/IEEE-CS Joint Task Force for Computing Curricula. Computing Curricula 2005: An Overview Report. Technical report, 2005. `http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf`, [Online; accessed 01-August-2013].

[65] ACM/IEEE-CS Joint Interim Review Task Force. Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force. Technical report, 2008. `http://www.acm.org/education/curricula/ComputerScience2008.pdf`, [Online; accessed 01-August-2013].

[66] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 30, pages 483–485, 1967.

[67] Herb Sutter. The Free Lunch is Over: a Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[68] Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7), September 2005.

[69] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*,

13(4), December 2003.
[70] Hans Muller and Kathy Walrath. Threads and Swing, September 2000. `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html` [Online; accessed 01-August-2013].
[71] The Go Community. The Go Language. `http://golang.org/`, 2013. [Online; accessed 01-August-2013].
[72] Zach Sims and Ryan Bubinski. Codecademy. `http://www.codecademy.com/`, 2013. [Online; accessed 01-August-2013].
[73] Nick Parlante. CodingBat Code Practice. `http://codingbat.com/`, 2013. [Online; accessed 01-August-2013].

A. The ProcessJ Home Page

We currently have a Web page in place at `http://www.processj.org`, the main page of which can be seen in Figure 6.

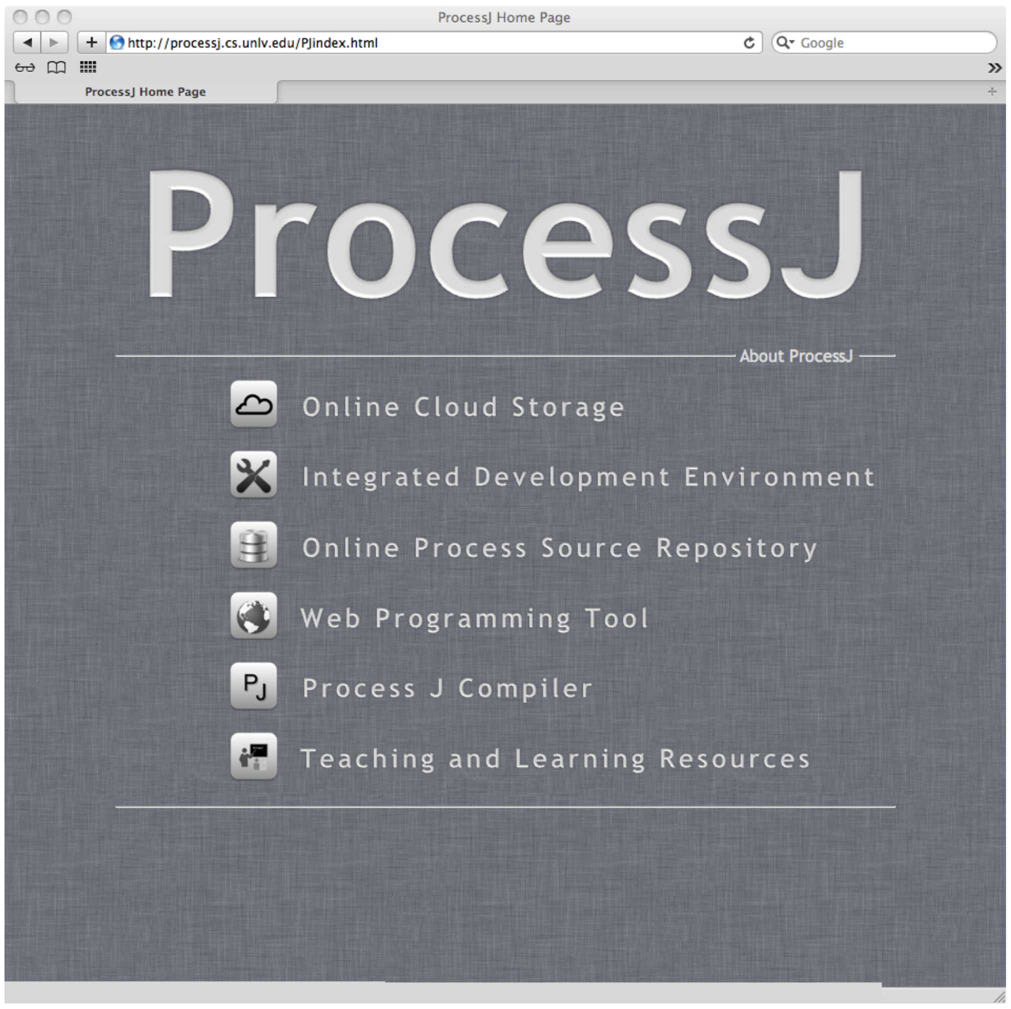

ProcessJ Home Page
http://processj.cs.unlv.edu/PJindex.html
Google
ProcessJ Home Page
ProcessJ
About ProcessJ
Online Cloud Storage
Integrated Development Environment
Online Process Source Repository
Web Programming Tool
Process J Compiler
Teaching and Learning Resources

Figure 6. The ProcessJ home page.