*Communicating Process Architectures 2006*
*Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)*
*IOS Press, 2006*



# Rain: A New Concurrent Process-Oriented Programming Language

Neil BROWN

*Computing Laboratory, University of Kent,*
*Canterbury, Kent, CT2 7NF, England.*
neil@twistedsquare.com

**Abstract.** This paper details the design of a new concurrent process-oriented programming language, Rain. The language borrows heavily from occam-π and C++ to create a new language based on process-oriented programming, marrying channel-based communication, a clear division between statement and expression, and elements of functional programming. An expressive yet simple type system, coupled with templates, underpins the language.

Modern features such as Unicode support and 64-bit integers are included from the outset, and new ideas involving permissions and coding standards are also proposed. The language targets a new virtual machine, which is detailed in a companion paper along with benchmarks of its performance.

**Keywords.** Process-oriented programming, Concurrency, Language design, Rain

## Introduction

Historically, desktop computing has been completely dominated by single-CPU, single-core machines. This is now changing — Intel and AMD, the two giants of desktop processor manufacture, both have a multi-core processor as their central market offering. It appears that the new dawn of parallelism has finally arrived, forced by the slowdown in the exponential growth of processor clock speeds; the race to increase the gigahertz has been replaced by a race to increase the cores.

Programming languages have not yet caught up to this shift. C and C++, still two of the most popular mainstream languages, completely lack any support for concurrency at the language level. Java has threads and monitors built-in to the language but using these for practical safe concurrency is not easy. The primary language with strong safe support for concurrency built-in is occam-π [2], a very different language to the C/C++/Java triumvirate.

Despite many innovations and developments, the level of abstraction of programming languages has moved at a glacial pace over the past sixty years. Broadly speaking, the progression has been: machine code, assembly language, imperative procedural languages (e.g. FORTRAN, C), and object-oriented languages (e.g. C++ [1], Java). It is my hope that the next step in that chain will be process-oriented programming. History suggests that in order for this to happen, the change will have to be made in small increments.

The gap between, say, Java and occam-π is vast in every respect. occam-π has no objects, collection classes (or built-in collection data types) or references and has a totally different syntax. occam-π encourages parallel code and makes use of channels for communicating between processes, rather than method calls between objects. From a practical business per-

[1] Although C++ is technically a multi-paradigm language, by far its most common mode of use is, in effect, object-oriented C

spective, the differences mean that re-training would be necessary and much existing/legacy code would have to be re-written.

Libraries such as JCSP [3], CTJ [4] and C++CSP [5] offer a bridge between current popular languages and fully process-oriented programming. However, these libraries suffer from the limitations of the language they are written in. Despite all the efforts of the library developers, programmers will always be able to write unsafe code using these libraries. For example, two C++CSP processes can both have a pointer to the same data structure which they can freely both modify concurrently, each overwriting the changes of the other. Such problems must be addressed at the language-level if they are to be eliminated.

Groovy Parallel [6] is an example of a project that helps to bridge the gap between mainstream languages and easy process-oriented programming. It is compatible with Java at the byte-code level. However, it still uses the Groovy language — unaltered — as its base. This means that the problems described above with JCSP et al. apply to Groovy Parallel. Honeysuckle [7] does solve these problems at the language level but is headed in a novel direction that diverges from some of the central concepts of process-oriented programming (such as channel communication).

I propose the development of a new process-oriented programming language, Rain. The language can be used on its own but will also be able to interface with C++CSP. This will allow existing C++ code to be used together with Rain code, with channels linking the C++CSP and Rain processes. The language will build on process-oriented programming and add new features such as templates and permissions (described later in this paper). The design of Rain is detailed in the remainder of this paper.

The Rain language is intended to follow the write-once run-anywhere pattern of Java and other interpreted languages. This will allow it to take advantage of heterogeneous concurrency mechanisms across multiple architectures without any code changes. This is described in detail in the accompanying paper about the Rain Virtual Machine (VM) [1], which also describes the C++ interface. The paper also provides performance benchmarks.

## 1. The Role of the Compiler

Studies carried out on the process of programming have shown that the earlier in the development/release cycle that a bug is caught, the less the cost to fix it [8]. Even if a test-first methodology is used, the effective development flow for the actual code is of the form: editor, compiler, unit-tests, system tests. Language-aware editors, usually present in Integrated Development Environments (IDEs) such as Eclipse [9], can help to highlight syntax errors before compilation takes place. The compiler can then spot some errors before transforming the code, and running it through the unit-tests which should (hopefully) catch any semantic errors, assuming that coverage is complete — which it rarely is, due to the difficulty of achieving complete coverage.

Different compilers can detect a widely different range of errors. An assembly language compiler can spot syntax mistakes, but as long as the operation is a valid one for the CPU, no other errors will be issued. Compilers for languages such as C++ and Java can pick up type errors and errors such as the potential use of a variable before initialisation. At the other end of the scale, interpreted languages have no compiler to speak of (other than the syntax-checking when loading a source file) and hence the compiler will also issue no errors. Errors such as type errors, unsupported interface methods and function/variable name typos, all picked up by the compiler in a language such as Java, will be detected by the tests. If the test coverage is incomplete then the errors will remain in the final code.

Given that the compiler is at such an early stage in development, and can be made powerful enough to detect a large proportion of common programming errors, it seems wise to

do just that. The compiler should eliminate as many potential errors as it can. This will need to be a combination of language design and compiler implementation.

In a presentation [10], Tim Sweeney of Epic Games (games programming typically being an area for the C/C++/Java/C# family of languages) provides a four-line C# function with five possible failures, followed by a semantically identical three-line pseudo-Haskell function with one possible failure. Solely by a combination of the language and the compiler, the same algorithm is made safer. Games programming has always been focused on the use of C-like languages for performance, but even in this domain it seems that higher-level, safer, expressive languages are seen as the future[2].

The Rain compiler will try to detect and issue compiler errors for as many potentially dangerous code patterns as possible. If the safety of a given piece of source code is unclear, in particular issues such as enforcing Concurrent-Read Exclusive-Write (CREW), the compiler should adopt a pessimistic (least-permissive) approach. Apart from this being the safest approach, in practical terms it is better for a future version of the compiler to accept a superset of the programs accepted by the current compiler rather than a subset — the latter option leading to non-compilable legacy code.

## 2. Processes and Functions

There are two major units of program code in Rain; processes and functions. A process is equivalent to a statement: it can affect and access external state (e.g. via channels), but it does not return a value explicitly — although using a channel it can accomplish a similar effect. A function is equivalent to an expression: it cannot affect or access external state (only its parameters) and will always return a value — otherwise it is a null statement from the caller's perspective. This means that functions can have no side-effects, and because they never depend on external state, only their passed parameters, they always have the potential to be inlined. Functions are permitted to be recursive.

It is hoped that this will allow a marriage of process-oriented and functional programming, with a cleaner syntax for the latter than occam-π. Although occam-π also contains this functions-as-expressions concept, it does not allow for recursive functions, nor the communication of functions (expanded on in section 4.4).

Functional languages have always had problems with any concept of I/O or time, due to their lack of state. This has made interaction with user interfaces or networks difficult — things that process-oriented languages can excel at. I/O can involve burdensome contortions such as Monads [11]. It is intended that the combination with process-oriented programming should remove such deficiencies of functional programming.

## 3. Communication

Like occam-π, Rain contains three communication primitives; channels, barriers and buckets. One-to-one unbuffered channels are available. Due to the anticipated potential implementation difficulties in the virtual machine no guarantees are offered regarding the availability buffers, any-to-one modality etc.

Like C++CSP and occam-π, Rain has the concept of both channel and channel end types. Channels cannot be used for anything except for accessing their channel ends. Channel ends may be used to read or write depending on the end they represent. There are some complications to this however: poisoning, `ALT`ing, and extended rendezvous.

[2] Although it is beyond the scope of this particular paper, Sweeney goes on to outline concurrency problems in game programming, which may well be of interest to the reader — including his opinion that "there is a wonderful correspondence between features that aid reliability and features that enable concurrency".

### 3.1. ALTing and Extended Rendezvous

ALTing, the term often used to describe that what East [7] terms selection (to distinguish from alternation), is a very powerful construct. Extended rendezvous (also referred to here as extended input) is an occam-π addition made by Barnes and Welch [12]. It essentially allows the reader to block the writer until the reader has completed further actions. This allows buffering/tapping processes to become essentially invisible to either end.

Not all channels that Rain is intended to support will be internal. Some channels may be networked, and some may be external channels such as `stdin` and `stdout` or plain sockets. Network channels may support extended rendezvous, but not ALTing. `stdin` may support ALTing, but extended input would not be possible (or applicable). This leads to the idea of channel-ends supporting a particular operation. In object-oriented programming this would be construed as a channel supporting a particular interface.

Some processes will require ALTing and/or extended rendezvous on their channels, and some will be indifferent to the support. For example, an extended-id process (that behaves like id, but with an extended input) will naturally require its reading end to support extended input, but will be indifferent to ALTing support on the channel. A two-channel merger process (that takes input on either of its two reading channels and sends it out on one output channel) would require ALTingbut not extended input.

Rain includes the concept of different channel input ends; the programmer can specify that a channel-reading end must support extended rendezvous or ALTing. If no such specifiers are included, such support is presumed not to be present. That is, by default the channel-reading end type (e.g. `?int`) is assumed to not support anything other than normal input.

### 3.2. Poisoning

The Communicating Process Architectures (CPA) community seems to be divided over the matter of stateful poisoning of channels [5,13,14,15]. Having seen its utility in programming with C++CSP, I chose to include it in Rain.

At the suggestion of Peter Welch, C++CSP was modified to include channel ends that were non-poisonable. That way, programs could hand out channel ends that could not be used to poison the rest of the network; for example the writing ends of an any-to-one channel that fed into an important server process. While any-to-one channels are currently not featured in Rain, the general logic prevails, and non-poisonable channel ends are explained below.

The idea of interfaces for channels could potentially be extended to distinguish between poisonable and non-poisonable channel ends. Consider the process id — there would need to be a version for poisonable channels and one for without:

```
process id_int
  (?int:in, !int:out)
{

  int : x;
  while (true)
  {
    in ? x;
    out ! x;
  }



}
```

```
process id_int_poison
  (poisonable ?int:in, poisonable !int:out)
{
  {
    int : x;
    while (true)
    {
      in ? x;
      out ! x;
    }
  } on poison {
    poison in;
    poison out;
  }
}
```

If only `in` was poisonable, and `out` were not, then another version would be required. Likewise if it were reversed. This approach would clearly be untenable. The solution is therefore that channel ends are assumed to be poisonable. Poisoning a non-poisonable channel end simply has no effect. Note that non-poisonable channel ends can still become poisoned by the process at the other end. So the `id_int_poison` process above would be the correct implementation (less the `poisonable` key-words) in order to catch the poison exceptions.

## 4. Types

Rain is strongly statically typed. Dynamic typing saves initial programmer effort, but experience has shown that this usually returns to haunt them in the form of run-time type errors. In line with the earlier discussion on the role of the compiler in section 1, it is preferred that the compiler do more work to save later problems.

Variables must be declared before use, with a specific type. They can be declared as constant. All function and process parameters are considered constant for their entire scope in the function/process. This saves confusion caused by reassigning function parameters. Descriptions of the types can be found in the following sub-sections.

### *4.1. Primitive Data Types*

Currently, mass-market computing is undergoing a transition from 32-bit to 64-bit architecture. Therefore, Rain includes 64-bit integers. I anticipate that larger integers will not be necessary in future. While similar phrases (usually foolish in retrospect) have been uttered in computing over the years, I am willing to state that I believe that 64-bit integers should be enough for most uses[3].

To avoid the horrid conventions now present in other languages (for example, the `long long` in C++), integers are labelled with their size and signed modality. So the full set of integers are: `sint8`, `sint16`, `sint32`, `sint64`, `uint8`, `uint16`, `uint32`, `uint64`.

`int` is also a built-in data type, and rather than adopt a sliding scale as per C/C++, it is defined to be an `sint64`. It is assumed that `int` will be used for most data (given that it will soon be the default word size on most machines), and the other types will be used when dealing with data that must be written on a network or to disk and hence requires a specific size.

The language currently offers `real32` and `real64` floating-point types. It is anticipated that the 128-bit version will be added in the future.

### *4.2. Communication Data Types*

Rain contains channels, channel-ends, barriers and buckets. Channels and channel-ends have a specified inner type, and channel-ends have a specified modality (input or output). Channels, barriers and buckets are all always constant. Making them constant prevents aliasing and also provides a clear point to allocate (upon declaration) and deallocate (at the end of its scope).

[3]For observed quantities larger than $2^{64}$ the purpose is likely scientific computing, which will probably be using floating-point numbers and/or a natively compiled language. The other common use of integers in computing is to store exact frequencies (having an incremented unique identifier in databases is effectively the same thing). Even if a new event occurred every nanosecond, it would still take over 500 years to overflow the 64-bit integer.

### *4.3. Complex Data Types*

#### *4.3.1. Named Tuples*

Tuples in languages such as Prolog suffer from a lack of scalability. From experience, programming with a nine-member tuple and always remembering the right field order (in a dynamically typed language, to add to the problem) is a difficult task.

Rain therefore offers what are referred to here as named tuples; a combination of records and tuples, almost a return to C structs:

```
typedef (real:x,real:y,real:z) : Point;
Point: p;
p = (0,1,1);
p.x = 1;
p = Point(z:0,y:0,x:0);
```

The tuples can be used either as tuples, or by named member access, or a combination of both in manner similar to Visual Basic's named parameter list.

#### *4.3.2. Lists*

C++ and Java did not build their main (non-array) list types into the language, but rather provided the language with tools from which the standard collection libraries were defined. This usually makes constructing these data structures difficult because there is no easy syntax. To define a list of the 3-D points (0,0,0) and (1,1,1) in Prolog would mean writing: `[(0,0,0) , (1,1,1)]`. In C++ the code would normally be along the lines of:

```
vector<Point> v;
v.push_back(Point(0,0,0));
v.push_back(Point(1,1,1));
```

Even accounting for the extra type information, this is an awkward way of creating a list. Only the fiendishly clever Boost [16] 'assign' libraries help alleviate this problem for arbitrarily sized-lists. Java 1.5 introduced a type-safe form of varargs to deal with this problem [17].

Rain also offers list types. In C, C++ and Java arrays and linked lists are two very different things; one was in-built and the other an object (or C's equivalent, a struct). Prolog only provides one list type, as does Python. Rain offers one list type, the underlying implementation of which (i.e. array or linked list) is guessed for best performance. Programmers may override this if desired.

```
typedef [Point] : PointList;
PointList: ps;
ps = [ (0,0,0) , (1,1,1) ];
ps += (2,2,2);
```

Lists support the addition operator (and therefore the += operator) but no other operators.

#### *4.3.3. Maps and Sets*

The two other commonly used data structures are maps (key/value structures) and sets. A set can be treated as a map with an empty value for each key. As such, only maps are built-in to Rain. Sets are provided by using an under-score as an empty data type. Maps support insertion/over-writing (through the assignment operator), removal, direct element access by key, presence checks and iteration.

Insertion/over-writing is guaranteed to always succeed. Removal of a non-present element is not considered an error and will have no effect. The only dangerous operation is element access (i.e. if the specified key is not present in the map) — this is covered in section 7 on exceptions.

```
typedef < int : Point > : NumberedPoints;
typedef <Point:_> : PointSet;
NumberedPoints : n;
PointSet : s;
s<(0,0,0)> = _;
remove s<(1,1,1)>;
n<3> = (1,0,1);
if (n has 4)
  s< n<4> > = _;
```

#### *4.3.4. Variants*

Named tuples, described above, are an example of product-type. Their complement is the sum-type, typified in a dangerous manner by C's union type; unions are dangerous because they do not keep a record of what their currently stored type is. Therefore a currently invalid type in a union can be accessed.

Many languages also supply types often referred to as enumerations. These are types consisting of a small closed set of values, identified by a meaningful name rather than merely a number. Their advantage over using integers with a set of constants is that the compiler can ensure the full set of values is handled in switch-like statements, and that the constants from two different types are not mixed (for example, a file error constant is not used in place of a GUI error constant).

In occam-π, similar ideas are combined to form variant protocols — typically an enumeration-like constant preceding a particular type, the result being somewhat similar to a safe union. In occam-π, variant protocols only exist in the form of a communicable type. In Rain, it is intended that these structures be a standard data type, usable wherever other data structures (lists, maps, etc) are. Haskell contains a similar concept of data with field labels. This idea requires further consideration before being finalised however.

### *4.4. Processes and Function Types*

Processes and functions can be stored and are therefore data types. Their type includes their parameter list (which itself is a named tuple type). Processes do not carry with them any state (which would be a much more complex mobility discussion [18]). In effect, an instance of process or function data is merely a code address. Therefore they can be assigned at will, but can never contain an invalid value (the compiler can mandate this).

Process and function data types can be transmitted over channels. This opens up interesting code patterns. Consider a filter process, with an incoming and outgoing channel of type `int`, and an incoming channel of a function that takes a single parameter of type `int` and returns a boolean. The filter process would accept an input on its channel of `int`, and send out the values if the function returns true — a dynamically configurable filter. The process would also accept input on its function channel, which would provide a new filter. Example code for the process is given overleaf (without poison handling for brevity):

```
process filter (?int : in,!int : out, ?function bool:(int) : filterIn)
{
  int: t;
  function bool:(int) : filter;
  filterIn ? filter;
  while (true)
  {
    pri alt
    {
      filterIn ? filter {}
      in ? t
      {
        if (filter(t))
          out ! t;
      }
    }
  }
}
```

### *4.5. Composing Data Types*

Data types in Rain are compositional, with a few necessary restrictions. Tuples and lists can contain any types. Maps can have any value type, but the key type must support ordering. Maps, communication primitives, processes and functions do not support ordering. Lists and tuples support ordering if (all of) their inner types do. All types support equality comparison.

Channels and channel-ends are a problem, as channel-ends must obey CREW. Consider the following code:

```
chan int: c;      #1
?int : in;        #2
[ ?int ] : a,b;   #3
in = c;           #4
a = [in];         #5
b = [in];         #6
seq all (x : a)   #7
    {b += x;}     #8
```

It would be relatively easy to spot the potential CREW problem (that is, the possibility of the non-shared channel end being used twice in parallel via the lists) on line 6. However, even if line 6 was removed, spotting the problem on same line 8 is harder. Situations of greater complexity are also imaginable. Therefore, adopting a least-permissive approach, channels and channel-ends cannot be contained in other data types (nor in other channels). For similar reasons, barriers and buckets cannot be contained in any other data types.

## 5. Iteration Constructs

Rain offers sequential and parallel iteration loops for lists and maps, in the following form:

```
seq all (x : list)
{
  ...  do something with x
}
```

```
par all ((key,value) : map)
{
  ...  do something with key and value
}
```

It is inherent in these constructs that all invalid-access problems are avoided (for example, array index out of bounds or map key not being present). These are therefore the preferred

forms of processing entire collections. Direct access of individual elements by index/key must be guarded by exception handlers as touched upon in section 7 on exceptions.

Note that the type of the iteration variables (`x`, `key` and `value`) is automatically deduced from the list/map type (where possible).

## 6. Templates

Consider the archetypal id process (shown without poison handling for brevity):

```
process id (?type : in, !type : out)
{
  type: x;
  while (true)
  {
    in ? x;
    out ! x;
  }
}
```

The only thing needed to compile this process is to know what `type` is. For a valid compilation, `type` could be any numeric type, a list, a map, or any other type that can be transmitted through channels. Having to write id over and over again for each type would be a nonsense. C++CSP is able to use templates to provide a single definition of a process — this is then compiled again whenever the process is used with a new type.

Experience has shown that templates are incredibly useful for creating libraries of common processes (thus encouraging code re-use and reducing programmer effort) and that they are generally a useful language feature. The type is substituted in at compile-time, not run-time, so it is just as safe as if each version were written out long-hand. This is a feature sadly lacking from any other process-oriented language that I am aware of.

In Rain, both processes and functions can be templated. Some form of compile-time reflection and/or partial specialisation is intended for inclusion, but the design of that is beyond the scope of this paper. Currently it will simply be plain type substitution. Either all types can be allowed (using the `any` keyword, or it can be restricted to numeric data types (using `numeric`). The conversion of id and successor are given below:

```
template (any: Type)
process id (?Type: in, !Type: out)
{
  Type: x;
  while (true)
  {
    in ? x;
    out ! x;
  }
}
```

```
template (numeric: Type)
process successor (?Type: in, !Type: out)
{
  Type: x;
  while (true)
  {
    in ? x;
    x += 1;
    out ! x;
  }
}
```

In C++, templates are provided by re-compiling the same code with the new type substituted in. This means that the source code is required (by including header files) every time the templated type is used. In Java, generics simply ‘auto-box’ the type, and thus the same piece of code is re-used for all instantiations of the generic object. C++ is able to use a templated type’s properties (e.g. making a method call on it) in a way that Java’s generics cannot without using inheritance. Rain adopts Java’s approach — through the use of type functions in the virtual machine (described in [1]), the equivalent of auto-boxing is performed.

## 7. Exceptions

Exceptions seem to have found favour in programming language design. C++, Java, the .NET languages and Python all include them. In their common implementation they allow error-handling to be collected in a single location, and for clean-up (certainly in the case of C++) to happen automatically during stack-unwinding. This is usually done to avoid checking for an error code on every call made. Below, the version on the left illustrates checking every call for an error, whereas the version on the right collects the error handling in the catch block.

```
if (file_open(...) == Error)
  { ... }
else if (file_write(...) == Error)
  { ... }
else if (file_close(...) == Error)
  { ... }
```

```
try {
  file_open (...);
  file_write (...);
  file_close (...);
} catch (FileException e)
  { ... }
```

C++CSP contained poison exceptions — thrown when an attempt was made to use a poisoned channel. As in the above example, exceptions were the best practical way of implementing stateful poisoning. The common poison-handling code for each process could be collected in one location.

There are a number of situations in Rain where the safety of an operation cannot be guaranteed at compile-time. The programmer usually has two alternatives: ensure that the operation will be safe with an appropriate check, or handle the exception. In the non-exception example below (on the left) the compiler can perform a simple static analysis and understand that the array access is safe. In the exception example on the right, an exception-handling block is provided.

```
if (xs.size > 5)
  { x = xs[5]; }
else
  { ... }
```

```
{
  x = xs[5];
} on invalid index
  { ... }
```

Exceptions, as in other languages, are a mechanism for collecting error handling. The currently intended exceptions in Rain are: invalid list indexing, invalid map access, poison and divide by zero.

The exceptions listed are required because of other features of the language. Most languages allow programmers to define, throw and catch their own exception types. This was a possibility for Rain. However, other languages have a strongly procedural basis that fits the perpetuation of thrown exceptions up the call stack. Rain allows functions, but they would definitely not be allowed to throw an exception (as a function call is only an expression). Processes would not be allowed to throw exceptions that can be caught by their parents. Exceptions are also not allowed to be caught outside a `par` block (due to the difficulties involved with parallel exceptions [14]). In Rain exceptions are a highly localised affair, and therefore do not have their procedural-nesting usefulness that they do in languages such as C++. Therefore the programmer is provided with no mechanism to define their own exceptions in Rain.

Error messages in process networks can either be perpetuated using error messages carried over channels or by using poison. It is up to the programmer which is used but the intention behind their design is thus. It is intended that poison be used only for unrecoverable errors and halting the program. For example, a process that is used to load up a file (the name of which is passed as a parameter) and send out its contents, should poison its channels if it discovers that the file does not exist. Without the file, the process has no purpose. By contrast, a process that has an incoming channel for file-names, and an outgoing channel for byte lists (of the file contents) should use error messages. If a requested file does not exist, the process

is still capable of continuing to function when processing the next requested file.

## 8. Text and Unicode

Unicode [19] was created to allow all known language characters to be stored as numbers, and yet still leave space for more in the standard. Unicode is available in a number of different encodings. Unicode prompts two considerations from the point of view of implementing a programming language; the compilation of source files, and support within the language itself for Unicode.

Support in the language itself is the trickier issue. Either a dedicated string type can be created and built-in to the language (as in Java), or some construction like a list/array of bytes (as in C) can be used. Given that a dedicated type would be stored as a list of bytes underneath the differences are really: how built-in to the language it should be, and what encodings should be offered/used by default.

Consider a program that takes a list of characters and sends them out one at a time on a channel. In ASCII, this would be done by accepting a list of `sint8`, and outputting a single `sint8` at a time. In UTF-8, this must be done by accepting a list of `sint8`, and outputting a list of `sint8`; one character can be multiple `sint8` bytes. Naturally, the temptation is to use an encoding where all characters are the same size. Technically, this is UTF-32, where every character is exactly four bytes. However, Unicode characters outside the two-byte range are quite rare.

Java originally picked a two-byte character size. At the time this was enough to hold all the planned Unicode characters [20]. Now that characters can be larger, the simplicity of having a character type that can hold all characters has been lost. Some believe that Java should resize its character class accordingly [21].

The decision of how large to make the default character type is therefore a trade-off between space efficiency (due to cache hits/misses, this also has an effect on performance) and the problems that would occur when encountering rare characters. The Unicode FAQ provides no specific guidance [22]. For simplicity of use, I have decided to use 32-bits as a character type. Strings, by default, will be a list of 32-bit values (therefore strictly one per character). The type `string` will be in-built, and exactly equivalent to `[uint32]`. Library functions and processes will be provided to aid conversion between encodings.

Rain source files are assumed to be UTF-8, although this will be made configurable via the command-line. White-space is used to separate variable names — therefore any non-white-space Unicode characters are valid in variable names. Any UTF-8 characters in a string literal between quotes `" "` will be converted and stored as UTF-32 for use as constant literals in the program .

## 9. Data Transmission

Concurrency has always been at odds with aliasing. Where aliasing is allowed in a program, two concurrent processes can have an alias for the same object, and CREW can be broken. Process-oriented languages have tried to prevent this by treating all data as private to the process. This presents an efficiency problem when large data structures are sent between processes. occam originally took the approach of always copying. occam-π introduced the concept of mobiles — essentially a non-aliasing reference [23]. Honeysuckle introduced the idea of ownership, a similar idea.

Mobiles, always-copy and duplicate-on-modify (allowing a reference to be shared as read-only, creating a new localised copy when it is modified) are three acceptable semantic solutions to the problem. The main question is whether to expose the dilemma to the

programmer, or hide the detail from them. Mobile data provides problems for the programmer (the potential for dereferencing an undefined mobile) and no benefits besides efficiency. Therefore including the idea of mobiles but hiding this detail from the programmer seems wise.

The compiler will use heuristics to decide which semantics to adopt as the underlying mechanism. From the programmer's perspective, it must appear as if always-copy is being used. Consider the following producer process:

```
seq all (int x : [0..100])
{
  [int]: list = [0..x];
  out ! list
}
```

The list is never referenced after its communication. Therefore in this example, mobile semantics would be wisest. Of course, the process on the other end of the communication needs to know whether it has received a reference that is duplicate-on-modify or not (from the receiver's perspective, a mobile reference is no different from a copied reference, as the receiver "owns" it). This involves a small amount of dynamic typing on the implementation side. Given that the programming language is being compiled to a virtual machine [1], the virtual machine can enforce the correct semantics.

## 10. Permissions

One of the challenges of programming is keeping on top of the design of a system. Even with one programmer this can be difficult; with multiple programmers and distributed teams it can become a nightmare. The compositional hierarchical designs that process-oriented programming naturally favours can help to alleviate this problem. Small components are easily composed into larger ones, and the interactions between the larger components (in the form of communication) are all visible from the parent component. However, if left unchecked, understanding which components are using external services can become difficult.

In a language such as Java, any part of a program can open up a socket or a file using the right API, and the calling method can be unaware of it. As a developer it has not been as uncommon an experience as it should be to dig into method call after method call, only to find that one of them is making an unexpected SQL query over a socket, or opening its own log file. In process-oriented programming, such activities can lead to unexpected deadlocks that are not obvious from a system overview.

Rain tentatively offers the idea of permissions (for want of a better name — services is ambiguous). A process must list the permissions that it requires. Potential examples include `network`, `file` or `gui`. Processes are permitted to pass on any or all of their permissions to their sub-processes, but permissions cannot be transferred (for example, via a channel). If process $P_A$ is the parent of $P_B$, which is the parent of $P_C$, and $P_C$ needs to write to a file, $P_B$ and $P_A$ will also need these permissions in order to pass them on down the hierarchy. This will allow it to be obvious from the top-level process in the program where the permissions are being granted, even if they are simply being granted everywhere (as is effectively the case in other languages).

## 11. Coding Standards

Coding standards have been a source of contention among programmers since they were conceived. Some believe that consistent coding standards for common patterns and naming

conventions are necessary on large projects to aid readability, lessen mistakes, and speed up programming when using code written by other developers. Others believe that they are a waste of time that only impedes programming.

Coding standards can be used because the standard-writer disagrees with a design decision taken by the language designer. For example, a C++ coding standard might disallow the choice operator (`a ? b : c`) because the standards-writer believes it unwisely hides choices in expressions.

Not all programmers will agree with the choices that have been made in Rain. They may prefer forcing variable declarations to be at the start of a block (as in C) rather than anywhere (as in C++/Java). Alternatively, they may want to avoid the use of exceptions as much as possible, and would therefore want to disbar explicit list-indexing and unguarded map accesses (favouring iteration constructs instead).

In the future, the compiler will support user-specified warnings and errors on normally-legal program code. The compiler will be able to take two types of inputs; source files, and (likely) XML files containing policy. Naturally the range of errors that can be supported will have to be restricted to relatively simple rules (mainly about syntax uses rather than semantic patterns), but this will allow the programmer to customise the compiler slightly to their wishes — as perverse as asking the compiler for more errors may seem. I expect that some Perl programmers will recognise this latter desire.

The idea of a seemingly-reconfigurable language may be alarming at first. The key detail is that these policies are always less permissive than the default. That is, the language with policies (coding standards) in place will always be a subset of the original language.

## 12. Implementation Progress

This paper has detailed the design of the Rain programming language. I have been implementing the compiler alongside the implementation of the virtual machine [1] that forms the target for the compiler. The framework and design of the compiler is complete, and a subset of most of the compiler stages has been implemented. Compilation of very simple programs is now possible. The compiler is targeted at the Windows and GNU/Linux platforms although I expect that it will compile on any operating system with a C++ compiler and a Boost [16] installation.

As with the virtual machine, the development has taken place on a mostly test-first basis, with copious unit tests. I believe that this provides a measure of assurance in the proper functioning of the compiler. Looking back, I do not believe I could have come as far without such testing, despite the extra time that it needed.

## 13. Conclusions and Future Work

This paper has presented the design of a new process-oriented programming language, Rain, intended for stand-alone use or integration with existing C++ code. Rain is statically typed with a rich set of data types and aims to eradicate as many errors as possible at compile-time. It offers (along with its VM) portable easy concurrency in a language that will interface well with C++CSP, thereby allowing it to interact with existing C++ libraries; in the future I particularly hope to include support for GUIs and networking, applications well suited to process-oriented programming.

Rain includes many innovations not found in other process-oriented programming languages. These include channel end interfaces (e.g. differentiating between channel end types that support `ALT`ing and those that do not), templates and permissions. These should make

the language interesting to current process-oriented programmers, who it is hoped will find these features useful.

It is unfortunate that further progress has not been made on the compiler; the implementation of the virtual machine alongside the compiler has meant that neither are yet finished. Future work will naturally involve, first and foremost, the completion of the compiler.

## Acknowledgements

I would like to thank Fred Barnes, Peter Welch and Ian East among many others for their outstanding work on occam-π and Honeysuckle. I hope they understand that if I step on their toes, it is only in an attempt to stand on their shoulders.

## Trademarks



## References

[1] N.C.C. Brown. Rain VM: Portable Concurrency through Managing Code. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, pages 253–267, September 2006.

[2] Fred Barnes. occam-pi: blending the best of CSP and the pi-calculus. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`, June 2006.

[3] University of Kent at Canterbury. Java Communicating Sequential Processes. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.

[4] Jan F. Broenink, Andrè W. P. Bakkers, and Gerald H. Hilderink. Communicating Threads for Java. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262, 1999.

[5] N.C.C. Brown and P.H. Welch. An Introduction to the Kent C++CSP Library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.

[6] Jon M. Kerridge. Groovy Parallel! A Return to the Spirit of occam? In Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood, editors, *Communicating Process Architectures 2005*, pages 13–28, September 2005.

[7] Ian R. East. The 'Honeysuckle' Programming Language: Event and Process. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 285–300, 2002.

[8] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[9] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, June 2006.

[10] Tim Sweeney. The Next Mainstream Programming Language: A Game Developer's Perspective 2006. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*. ACM, 2006. Available at: `http://www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt`, June 2006.

[11] Stefan Klinger. The Haskell Programmer's Guide to the IO Monad. Don't Panic. December 2005. Available at: `http://db.ewi.utwente.nl/Publications/PaperStore/db-utwente-0000003696.pdf`, June 2006.

[12] Fred Barnes and Peter Welch. Prioritised Dynamic Communicating Processes - Part I. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, 2002.

[13] Jan F. Broenink and Dusko S. Jovanovic. On Issues of Constructing an Exception Handling Mechanism for CSP-Based Process-Oriented Concurrent Software. In Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood, editors, *Communicating Process Architectures 2005*, pages 29–41, 2005.

[14] Gerald H. Hilderink. Exception Handling Mechanism in Communicating Threads for Java. In Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood, editors, *Communicating Process Architectures 2005*, pages 313–330, 2005.
[15] Correspondence on the occam-com mailing list, 2nd march 2006.
[16] Boost C++ Libraries. `http://www.boost.org/`, June 2006.
[17] Sun Microsystems Inc. Varargs. `http://java.sun.com/j2se/1.5.0/docs/guide/language/varargs.html`, June 2006.
[18] F.R.M.Barnes and P.H.Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings-Software*, 150(2):121–136, April 2003.
[19] Unicode Inc. Unicode Home Page. `http://www.unicode.org/`, June 2006.
[20] Sun Microsystems. Class Character (Java 2 Platform SE 5.0). `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Character.html`, June 2006.
[21] ONJava. 10 Reasons We Need Java 3.0. `http://www.onjava.com/pub/a/onjava/2002/07/31/java3.html`, June 2006.
[22] Unicode Inc. Unicode FAQ — Programming Issues. `http://www.unicode.org/faq/programming.html`, June 2006.
[23] F.R.M.Barnes and P.H.Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In *Communicating Process Architectures*. IOS Press, 2001.