CSP as a Domain-Specific Language Embedded in Python and Jython

Sarah MOUNT [1], Mohammad HAMMOUDEH, Sam WILSON and Robert NEWMAN

School of Computing and I.T., University of Wolverhampton, U.K.

Abstract. Recently, much discussion has taken place within the Python programming community on how best to support concurrent programming. This paper describes a new Python library, `python-csp`, which implements synchronous, message-passing concurrency based on Hoare's Communicating Sequential Processes. Although other CSP libraries have been written for Python, `python-csp` has a number of novel features. The library is implemented both as an object hierarchy and as a domain-specific language, meaning that programmers can compose processes and guards using infix operators, similar to the original CSP syntax. The language design is intended to be idiomatic Python and is therefore quite different to other CSP libraries. `python-csp` targets the CPython interpreter and has variants which reify CSP process as Python threads and operating system processes. An equivalent library targets the Jython interpreter, where CSP processes are reified as Java threads. `jython-csp` also has Java wrappers which allow the library to be used from pure Java programs. We describe these aspects of `python-csp`, together with performance benchmarks and a formal analysis of channel synchronisation and choice, using the model checker SPIN.

Keywords. CSP, domain-specific languages, dynamic languages, Python

Introduction

Python is a lexically scoped, dynamically typed language with object-oriented features, whose popularity is often said to be due to its ease of use. The rise of multi-core processor architectures and web applications has turned attention in the Python community to concurrency and distributed computing. Recent versions of Python have language-level or standard library support for coroutines[2], (system) threads[3] and process[4] management, the latter two largely in the style of the POSIX thread library. This proliferation of concurrency styles is somewhat in contrast to the "Zen" of Python [1] which states that "There should be one—and preferably only one—obvious way to do it.".

One reason for adding coroutines (and therefore the ability to use "green" threads) and operating-system processes is the performance penalty of using Python threads, which is largely due to the presence of the global interpreter lock (GIL) in the C implementation of the Python interpreter. The GIL is implemented as an operating system semaphore or condition variable which is acquired and released in the interpreter every time the running thread blocks

---

[1] Corresponding Author: *Sarah Mount, School of Computing and I.T., University of Wolverhampton, Wulfruna St., Wolverhampton, WV1 1SB, U.K.*. Tel.: +44 1902 321832; Fax: +44 1902 321478; E-mail: `s.mount@wlv.ac.uk`.

[2] Python PEP 342: Coroutines via Enhanced Generators `http://www.python.org/dev/peps/pep-0342/`

[3] `http://docs.python.org/library/threading.html#module-threading`

[4] `http://docs.python.org/library/multiprocessing.html#module-multiprocessing`

for I/O, allowing the operating system to schedule a different thread. A recent presentation by Dave Beazley[5] contained benchmarks of the following CPU-bound task:

```
def count(n):
    while n > 0:
        n -= 1
```

and found that a parallel execution of the task in threads performed 1.8 times slower than a sequential execution and that performance improved if one (of two) CPU cores was disabled. These counter-intuitive results are often the basis for developers to call for the GIL to be removed. The Python FAQ[6] summarises why the GIL is unlikely to be removed from the reference implementation of the interpreter, essentially because alternative implementations of thread scheduling have caused a performance penalty to single-threaded programs. The current solution, therefore, is to provide programmers with alternatives: either to write single-threaded code, perhaps using coroutines for cooperative multitasking, or to take advantage of multiple cores and use processes and IPC in favour of threads and shared state. A second solution is to use another implementation of Python, apart from the CPython interpreter. Stackless Python [2] is an implementation which largely avoids the use of the C stack and has green threads (called "tasklets") as part of its standard library. Google's Unladen Swallow[7] is still in the design phase, but aims to improve on the performance of CPython five-fold and intends to eliminate the GIL in its own implementation by 2010.

This paper describes another alternative, to augment Python with a higher-level abstraction for message-passing concurrency, `python-csp` based on Hoare's Communicating Sequential Processes [3]. The semantics of CSP are relatively abstract compared with libraries such as pthreads and so the underlying implementation of CSP "processes" as either system threads, processes or coroutines is hidden from the user. This means that the user can choose an implementation which is suitable for the interpreter in use or the context of the application they are developing (for example, processes where a multi-core architecture is expected to be used, threads where one is not). Also CSP was designed specifically to help avoid well-known problems with models such as shared memory concurrency (such as deadlocks and race conditions) and to admit formal reasoning. Both properties assist the user by encouraging program correctness. The authors have a particular interest in using Python to implement complex tasks requiring coordination between many hosts. The SenSor simulator and development tool [4,5] provided facilities for prototyping algorithms and applications for wireless sensor networks in pure Python, using shared-memory concurrency. The burden of managing explicit locking in an already complex environment made the implementation of SenSor difficult. A new version of the tool is currently under development and will be built on the `python-csp` and `jython-csp` libraries.

To deal with the different performance characteristics of threads and processes in the current implementations of Python, the `python-csp` library currently has a number of different implementations:

- The `csp.cspprocess` module which contains an implementation of `python-csp` based on operating system processes, as managed by the `multiprocessing` library; and
- the `csp.cspthread` module which contains an implementation of `python-csp` based on system threads.

There is also a version of the library called `jython-csp` that targets Jython, a version of Python which runs on the Java VM. `jython-csp` uses Java threads (which are also sys-

[5] http://blip.tv/file/2232410
[6] http://www.python.org/doc/faq/library/
[7] http://code.google.com/p/unladenswallow/

tem threads), rather than Python threads. Jython allows the user to mix Java and Python code in programs (almost) freely. As such, `jython-csp` allows users to use any combination of Python and Java, including being able to write pure Java programs using wrappers for `jython-csp` types. In general, in this paper we will refer to `python-csp`, however, the syntax and semantics of the libraries can be assumed to be identical, unless stated otherwise.

The remainder of this paper describes the design and implementation of `python-csp` and `jython-csp`. Section 1 gives an overview of the design philosophy of the library and its syntax and (informal) semantics. Section 2 describes a longer `python-csp` program and gives a discussion of the design patterns used in the implementation. Section 3 begins an evaluation of `python-csp` by describing benchmark results using the Commstime program and comparing our work with similar libraries. `python-csp` and `jython-csp` have been bench-marked against PyCSP [6], another realisation of CSP in Python and JCSP [7], a Java library. Section 4 outlines ongoing work on model checking channel synchronisation and non-deterministic selection in the `python-csp` implementation. Section 5 concludes and describes future work.

1. python-csp and jython-csp: Syntax and Semantics

The design philosophy behind `python-csp` is to keep the syntax of the library as "Pythonic" and familiar to Python programmers as possible. In particular, two things distinguish this library from others such as JCSP [7] and PyCSP [6]. Where languages such as Java have strong typing and sophisticated control over encapsulation, Python has a dynamic type system, often using so-called "duck typing" (which means that an object is said to implement a particular type if it shares enough data and operations with the type to be used in the same context as the type). Where an author of a Java library might expect users to rely on the compiler to warn of semantic errors in the type-checking phase, Python libraries tend to trust the user to manage their own encapsulation and use run-time type checking. Although Python is a dynamically typed language, the language is helpful in that few, if any, type coercions are implicit. For example, where in Java, a programmer could concatenate a `String` and an `int` type when calling `System.out.println`, the equivalent expression in Python would raise an exception. In general, the Python type system is consistent and this is largely because every Python type is reified as an object. Java differs from Python in this respect, as primitive Java types (`byte`, `short`, `int`, `long`, `char`, `float`, `double`, `boolean`) do not have fields and are not created on the heap. In Python, however, all values are (first-class) objects, including functions and classes. Importantly for the work described here, operators may be overloaded for new types as each Python operator has an equivalent method inherited from the base object, for example:

```
>>> 1 + 2
3
>>> (1).__add__(2)
3
>>> [1] * 3
[1, 1, 1]
>>> ([1]).__mul__(3)
[1, 1, 1]
>>>
```

Lastly, Python comes with a number of features familiar to users of functional programming languages such as ML that are becoming common in modern, dynamically typed languages. These include generators, list comprehensions and higher-order functions.

CSP [3] contains three fundamental concepts: processes, (synchronous) channel communication and non-deterministic choice. `python-csp` provides two ways in which the user may create and use these CSP object types: one method where the user explicitly creates

instances of types defined in the library and calls the methods of those types to make use of them; and another where users may use syntactic sugar implemented by overriding the Python built in infix operators. Operator overloading has been designed to be as close to the original CSP syntax as possible and is as follows:

Syntax	Meaning	CSP equivalent
$P > Q$	Sequential composition of processes	$P\,;\,Q$
$P \& Q$	Parallel composition of processes	$P \,\|\|\, Q$
$c_1 \mid c_2$	Non-deterministic choice	$c_1 \sqcap c_2$
$n * A$	Repetition	$n \bullet A$
$A * n$	Repetition	$n \bullet A$
$Skip()$	Skip guard, always ready to synchronise	$Skip$

where:

- n is an integer;
- P and Q are processes;
- A is a non-deterministic choice (or ALT); and
- c_1 and c_2 are channels.

The following sections describe each of the python-csp features in turn.

1.1. python-csp Processes

In python-csp a process can be created in two ways: either explicitly by creating an instance of the CSPProcess class or, more commonly, by using the @process decorator[8]. In either case, a callable object (usually a function) must be created that describes the run-time behaviour of the process. Listing 1 shows the two ways to create a new process, in this case one which opens a sensor connected to the USB bus of the host and continuously prints out a transducer reading every five minutes. Whichever method is used to create the process, P, a special keyword argument _process must be passed in with a default value. When the process P is started (by calling its start method) _process is dynamically bound to an object representing the underlying system thread or process which is the reification of the CSPProcess instance. This gives the programmer access to values such as the process identifier (PID), or thread name which may be useful for logging and debugging purposes. When the start methods in a CSPProcess object has returned the underlying thread or process will have terminated. Once this has happened, accessing the methods or data of the corresponding _process variable will raise an exception.

```
# Using the CSPProcess class:
def print_rh_reading():
    rhsensor = ToradexRH() # Oak temp / humidity sensor
    rhsensor.open()
    while True:
        data = rhsensor.get_data()
        print 'Humidity %%g: Temp: %gC' % data[1:]
        dingo.platform.gumstix.sleep(60 * 5) # 5 min
P = CSPProcess(print_rh_reading, _process=None)
P.start()

# Using the @process decorator:
```

[8] A "decorator" in Python is a callable object "wrapped" around another callable. For example, the definition of a function fun decorated with the @mydec decorator will be replaced with fun = mydec(fun).

```
@process
def print_rh_reading(_process=None):
    rhsensor = ToradexRH() # Oak temp / humidity sensor
    rhsensor.open()
    while True:
        data = rhsensor.get_data()
        print 'Humidity %%g: Temp: %gC' % data[1:]
        dingo.platform.gumstix.sleep(60 * 5) # 5 min
P = print_rh_reading()
P.start()
```

Listing 1. Two ways to create a python-csp process.

1.2. python-csp Parallel and Sequential Execution

CSP processes can be composed either sequentially or in parallel. In sequential execution each process starts and terminates before the next in the sequence begins. In parallel execution all processes run “at once” and therefore the order of any output they effect cannot be guaranteed. Parallel and sequential execution can be implemented in `python-csp` either by instantiating `Par` and `Seq` objects or by using the overloaded `&` or `>` operators. In general, using the overloaded infix operators results in clear, simple code where there are a small number of processes. Listing 2 demonstrates sequential and parallel process execution in `python-csp`.

```
... def P(n, _process=None):
...     print n
...
>>> # In sequence, using syntactic sugar...
>>> P(1) > P(2)
1
2
>>> # In sequence, using objects...
>>> Seq(P(1), P(2)).start()
1
2
>>> # In parallel, using syntactic sugar...
>>> P(1) & P(2)
2
1
>>> # In parallel, using objects...
>>> Par(P(1), P(2)).start()
1
2
```

Listing 2. Two ways to run processes sequentially and in parallel.

1.3. python-csp Channels

In CSP communication between processes is achieved via channels. A channel can be thought of as a pipe (similar to UNIX pipes) between processes. One process writes data down the channel and the other reads. Since channel communication in CSP is synchronous, the writing channel can be thought of as offering data which is only actually written to the channel when another process is ready to read it. This synchronisation is handled entirely by the language, meaning that details such as locking are invisible to the user. Listing 3 shows how channels can be used in `python-csp`.

In its csp.cspprocess implementation, python-csp uses an operating system pipe to transmit serialised data between processes. This has a resource constraint, as operating systems limit the number of file descriptors that each process may have open. This means that although python-csp programs can create any number of processes (until available memory is saturated), a limited number of channels can be created. In practice this is over 600 on a Ubuntu Linux PC. To compensate for this, python-csp offers a second implementation of channels called FileChannel. A FileChannel object behaves in exactly the same way as any other channel, except that it uses files on disk to store data being transferred between processes. Each read or write operation on a FileChannel opens *and* closes the operating system file, meaning that the file is not open for the duration of the application running time. Programmers can use FileChannel objects directly, or, if a new Channel object cannot be instantiated then Channel will instead return a FileChannel object.

A third class of channel is provided by python-csp, called a NetworkChannel. A NetworkChannel transfers data between processes via a socket listener which resides on each node in the network for the purpose of distributing channel data. By default when the Python csp.cspprocess package is imported, a socket server is started on the host (if one is not already running).

```
@process
def send_rh_reading(cout, _process=None):
    rhsensor = ToradexRH() # Oak temp / humidity sensor
    timer = TimerGuard()
    rhsensor.open()
    while True:
        data = rhsensor.get_data()
        cout.send('Humidity %%g: Temp: %gC' % data[1:])
        timer.sleep(60 * 5) # 5 min
        guard.read()        # Synchronise with timer guard.

@process
def simple_sensor(_process=None):
    ch = Channel()
    Printer(ch) & send_rh_reading(ch)
```

Listing 3. Two processes communicating via a channel.

1.4. Non-Deterministic Selection and Guards in python-csp

Non-deterministic selection (called "select" in JCSP and "ALT" or "ALTing" in occam) is an important feature of many process algebras. Select allows a process to choose between a number of waiting channel reads, timers or other "guards" which are ready to synchronise. In python-csp this is achieved via the select method of any guard type. To use select an Alt object must be created and should be passed any number of Guard instances. See Listing 4 for an example. The select method can be called on the Alt object and will return the value returned from the selected guard. To implement a new guard type, users need to subclass the Guard class and provide the following methods:

enable which should attempt to synchronise.
disable which should roll back from an enable call to the previous state.
is_selectable which should return True if the guard is able to complete a synchronous transaction and False otherwise.
select which should complete the synchronous transaction and return the result (note this semantics is slightly different from that found in JCSP, as described by Welch [7], where an index to the selected guard in the guard array is returned by select).

`poison` which should be used to finalize and delete the current guard and gracefully terminate any processes connected with it [8,9].

Since Python has little support for encapsulation, the list of guards inside an `Alt` object is available to any code which has access to the `Alt`.

```
@process
def printAvailableReading(cins, _process=None):
    alt = Alt(cins)
    while True:
        print alt.select()
@process
def simpleSensorArray(_process=None):
    chans, procs = [], []
    for i in NUMSENSORS:
        chans.append(Channel())
        procs.append(sendRHReading(chans[-1]))
    procs.append(printAvailableReading(chans))
    Par(*procs).start()
```

Listing 4. Servicing the next available sensor reading with non-deterministic selection.

Like many other implementations of CSP, `python-csp` implements a number of variants of non-deterministic selection:

`select` enables all guards and either returns the result of calling `select` on the first available guard (if only one becomes available) or randomly chooses an available guard and returns the result of its `select` method. The choice is truly random and determined by a random number generator seeded by the `urandom` device of the host machine.

`priority_select` enables all guards and, if only one guard becomes available then `priority_select` returns the result of its `select` method. If more than one guard becomes available the first guard in the list passed to `Alt` is selected.

`fair_select` enables all guards and, if only one guard becomes available then `fair_select` returns the result of its `select` method. `Alt` objects keep a reference to the guard which was selected on the previous invocation of any `select` method, if there has been such an invocation and the guard is still in the `guards` list. If `fair_select` is called and more than one guard becomes available, then `fair_select` gives lowest priority to the guard which was returned on the previous invocation of any of the select methods. This idiom is used to reduce the likelihood of starvation as every guard is guaranteed that no other guard will be serviced twice before it is selected.

There are two forms of syntactic sugar that `python-csp` implements to assist in dealing with `Alt` objects: a choice operator and a repetition operator. Using the choice operator, users may write:

```
result = guard_1 | guard_2 | ... | guard_n
```

which is equivalent to:

```
alt = Alt(guard_1, guard_2, ..., guard_n)
result = alt.select()
```

To repeatedly select from an `Alt` object n times, users may write:

```
gen = n * Alt(guard_1, guard_2, ..., guard_n)
```

or:

```
gen = Alt(guard_1, guard_2, ..., guard_n) * n
```

this construct returns a generator object which can be iterated over to obtain results from `Alt.select` method calls. Using generators in this way is idiomatic in Python and will be familiar to users. The following is a typical use case:

```
gen = Alt(guard_1, guard_2, ..., guard_n) * n
while True:
    ... gen.next() ...
```

Each time `gen.next()` is called within the loop, the `select()` method of the `Alt` object is called, and its result returned.

In addition to channel types, `python-csp` implements two commonly used guard types: `Skip` and `TimerGuards`. `Skip` is the guard which is always ready to synchronise. In `python-csp` its `select` method always returns `None`, which is the Python null value. `TimerGuards` are used to either suspend a running process (by calling their `sleep` method) or as part of a synchronisation where the guard will become selectable after a timer has expired:

```
@process
def alarm(self, cout, _process=None):
    alt = Alt(TimerGuard())
    t0 = alt.guard[0].read()  # Fetch current time
    alt.guard[0].set_alarm(5) # Selectable 5 secs from now
    alt.select()
    duration = guard.read() - t0 # In seconds
    cout.write(duration)
```

1.5. Graceful Process Termination in python-csp

Terminating a parallel program without leaving processes running in deadlock is a difficult problem. The most widely implemented solution to this problem was invented by Peter Welch [10] and is known as "channel poisoning". The basic idea is to send a special value down a channel which, when read by a process, is then propagated down any other channels known to that process before it terminates. In `python-csp` this can be affected by calling the `poison` method on any guard.

A common idiom in `python-csp`, especially where producer-consumer patterns are implemented, is this:

```
alt = Alt(*channels)
for i in xrange(len(channels)):
    alt.select()
```

Here, it is intended that each guard in `channels` be selected exactly once. Once a guard has been selected its associated writer process(es) will have finished its computation and terminate. In order to support this idiom efficiently, `python-csp` implements a method called `poison` on `Alt` objects which serves to poison the writer process(es) attached to the last selected guard and remove that guard from the list, used as follows:

```
a = Alt(*channels)
for i in xrange(len(channels)):
    a.select()
    a.poison()
```

By shortening the list of guards less synchronisation is required on each iteration of the `for` loop, reducing the computational effort required by the `select` method.

1.6. Built-In Processes and Channels

`python-csp` comes with a number of built-in processes and channels, aimed to speed development. This includes all of the names defined in the JCSP "plugnplay" library. In addition to these and other built-in processes and guards, `python-csp` comes with analogues of every unary and binary operator in Python. For example, the `Plus` process reads two values from channels and then writes the addition of those values to an output channel. An example implementation[9] of this might be:

```
@process
def Plus(cin1, cin2, cout, _process=None):
    while True:
        in1 = cin1.read()
        in2 = cin2.read()
        cout.write(in1 + in2)
    return
```

Listing 5. A simple definition of the built-in `Plus` process.

1.7. jython-csp Implementation and Integration with Pure Java

`jython-csp` is a development of `python-csp` for integration with Jython. Jython has similar semantics as Python but uses the Java runtime environment (JRE) which allows access to the large number of Java libraries, such as Swing, which are useful, platform-independent and well optimised. `jython-csp` has similar workings to the initial `python-csp` with the ability to utilise any class from the standard Java and Python class libraries. `jython-csp` utilises Java threads and would be expected to perform similarly to other CSP implementations based on Java threads, such as JCSP (e.g. [7]). A comparison between `jython-csp` and `python-csp` implementations of the Monte Carlo approximation to π is shown in Table 1.

Table 1. Results of testing Java threads against Python threads and OS processes.

Thread Library	Running time of π approximation (seconds)
`jython-csp` (Java Threads)	26.49
`python-csp` (Python Threads)	12.08
`python-csp` (OS Processes)	9.59

As we shall see in Section 3 the JCSP library performs channel communication very efficiently, so one might expect that `jython-csp` would also execute quickly. A speculation as to why `jython-csp` (using Java threads) performs poorly compared to the other CSP implementations is a slow Java method dispatch within Jython.

In addition to the modification of the threading library used `jython-csp` also takes advantage of Java locks and semaphores from the `java.util.concurrent` package. `jython-csp` has no dependency on non standard packages; the library will work with any JRE which is compatible with Jython 2.5.0final.

[9] In fact, these processes are implemented slightly differently, taking advantage of Python's support for reflection. Generic functions called `_applybinop` and `_applyunaryop` are implemented, then `Plus` may be defined simply as `Plus = _applybinop(operator.__add__)`. The production version of this code is slightly more complex as it allows for documentation for each process to be provided whenever `_applybinop` and `_applyunaryop` are called.

1.8. Java-csp

Java-csp is the integration of `jython-csp` with Java to allow Java applications to utilise the flexibility of `jython-csp`.

The Java CSP implementation attempts to emulate the built in Java thread library (`java.lang.Thread`) with a familiar API. As with Java threads there are two ways to use threads in an application:

- By extending the Thread class and overwriting the run method; or
- By implementing the Runnable interface.

The Java csp implementation has a similar usage:

- Extending the `JavaCspProcess` and overwriting the `target` method; or
- Implementing the `JavaCspProcessInterface` interface.

`jython-csp` and `python-csp` uses the pickle library as a means of serialising data down a channel. Pickle takes a Python/Jython object and returns a sequence of bytes; this approach only works on Python/Jython object and is unsuitable for native Java objects. As a solution java-csp implements a wrapped version of Java object serialization which allows Jython to write pure Java objects, which implement the `Serializable` interface, to a channel, in addition to this, Python/Jython objects can be written down a channel if they extend the `PyObject` class.

2. Mandelbrot Generator: an Example python-csp Program

2.1. Mandelbrot Generator

Listing 6 shows a longer example of a `python-csp` program as an illustration of typical coding style. The program generates an image of a Mandelbrot set which is displayed on screen using the PyGame library[10] (typically used for implementing simple, 2D arcade games).

The Mandelbrot set is an interesting example, since it is embarrassingly parallel – i.e. each pixel of the set can be calculated independently of the others. However, calculating each pixel of a large image in a separate process may be a false economy. Since channel communication is likely to be an expensive part, the resulting code is likely to be I/O bound. The code in Listing 6 is structured in such a way that each column in the image is generated by a separate "producer" process. Columns are stored in memory as a list of RGB tuples which are then written to a channel shared by the individual producer process and a single "consumer" process. The main work of the consumer is to read each image column via an `Alt` object and write it to the display surface.

This producer-consumer architecture is common to many parallel and distributed programs. It it not necessarily, however, the most efficient structure for this program. Since some areas of the set are essentially flat and so simple to generate, many producer processes are likely to finish their computations early and spend much of their time waiting to be selected by the consumer. If this is the case, it may be better to use a "farmer" process to task a smaller number of "worker" processes with generating a small portion of the image, and then re-task each worker after it has communicated its results to the farmer. Practically, if efficiency is an important concern, these options need to be prototyped and carefully profiled in order to determine the most appropriate solution to a given problem.

```
from csp.cspprocess import *
@process
def mandelbrot(xcoord, (width, height), cout,
```

[10] `http://www.pygame.org`

```
                acorn=-2.0, bcorn=-1.250, _process=None):
    # Generate image data for column xcoord...
    cout.write((xcoord, imgcolumn))
    _process._terminate()
@process
def consume(IMSIZE, filename, cins, _process=None):
    # Create initial pixel data.
    pixmap = Numeric.zeros((IMSIZE[0], IMSIZE[1], 3))
    pygame.init()
    screen = pygame.display.set_mode((IMSIZE[0], IMSIZE[1]), 0)
    # Wait on channel data.
    alt = ALT(*cins)
    for i in range(len(cins)):
            xcoord, column = alt.select()
        alt.poison() # Remove last selected guard and producer.
        # Update column of blit buffer
        pixmap[xcoord] = column
        # Update image on screen.
        pygame.surfarray.blit_array(screen, pixmap)
def main(IMSIZE, filename):
    channels, processes = [], []
    for x in range(IMSIZE[0]): # Producer + channel per column.
        channels.append(Channel())
        processes.append(mandelbrot(x, IMSIZE, channels[x]))
    processes.insert(0, consume(IMSIZE, filename, channels))
    mandel = PAR(*processes)
    mandel.start()
```

Listing 6. Mandelbrot set in `python-csp` (abridged).

The worker-farmer architecture shows an improvement in performance over the producer-consumer architecture. The code in Listing 7 is structured in such a way that each column in the image is computed by a single process. Initially a set of workers are created and seeded with the value of the column number, the pixel data for that column is generated then written to a channel. When the data has been read, the “farmer” then assigns a new column for the worker to compute. If there are no remaining columns to be generated, the farmer will write a terminating value and the worker will terminate. This is required to instruct the “worker” that there are no more tasks to be performed. The consumer has the same function as before although with a smaller set of channels to choose from. Workers which have completed their assigned values have a shorter amount of time to wait until the `Alt` object selects them.

```
@process
def mandelbrot(xcoord, (width, height), cout,
                acorn=-2.0, bcorn=-1.250, _process=None):
    # Generate image data for column xcoord...
    cout.write((xcoord, imgcolumn))
    xcoord = cout.read()
    if xcoord == -1:
        _process._terminate()
@process
def consume(IMSIZE, filename, cins, _process=None):
    global SOFAR
    # Create initial pixel data
    pixmap = Numeric.zeros((IMSIZE[0], IMSIZE[1], 3))
    pygame.init()
    screen = pygame.display.set_mode((IMSIZE[0], IMSIZE[1]), 0)
    # Wait on channel data
    alt = Alt(*cins)
```

```
for i in range(IMSIZE[0]):
    xcoord, column = alt.pri_select()
    # Update column of blit buffer
    pixmap[xcoord] = column
    # Update image on screen.
    pygame.surfarray.blit_array(screen, pixmap)
    if SOFAR < IMSIZE[0]:
        alt.last_selected.write(SOFAR)
        SOFAR = SOFAR + 1
    else:
        alt.last_selected.write(-1)
```

Listing 7. Mandelbrot set in `python-csp` (abridged) using the "farmer" /"worker" architecture.

The improvement in performance can been seen in Figure 1: using a smaller number of processes reduces the run time of the program.

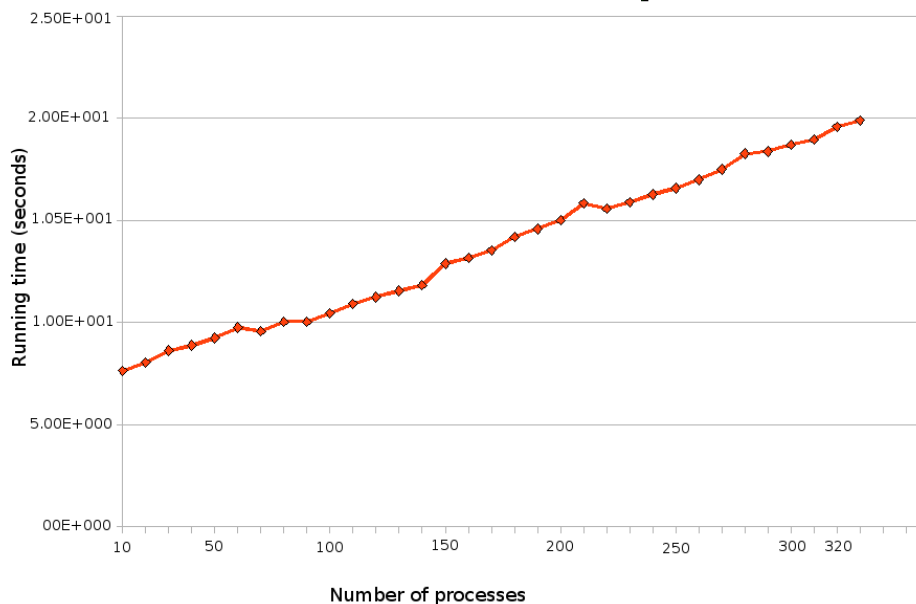

Figure 1. Run times of "farmer" / "worker" Mandelbrot program with different numbers of CSP processes.

The graph shows a linear characteristic, which would be expected as the select method in `Alt` is $O(n)$.

3. Performance Evaluation and Comparison to Related Work

The Commstime benchmark was originally implemented in occam by Peter Welch at the University of Kent at Canterbury and has since become the *de facto* benchmark for CSP implementations such as occam-π [11], JCSP [7] and PyCSP [6].

Table 2 shows results of running the Commstime benchmark on JCSP version 1.1rc4, PyCSP version 0.6.0 and `python-csp`. To obtain fair results the implementation of Commstime used in this study was taken directly from the PyCSP distribution, with only syntactic changes made to ensure that tests run correctly and are fairly comparable. In each case, the type of "process" used has been varied and the default channel implementation has been used. In the case of the `python-csp` channels are reified as UNIX pipes. The JCSP implementation uses the `One2OneChannelInt` class. The PyCSP version uses the default PyCSP `Channel` class for each process type. All tests were run on an Intel Pentium dual-core 1.73 GHz CPU and

1 GB RAM, running Ubuntu 9.04 (Jaunty Jackalope) with Linux kernel 2.6.28-11-generic. Version 2.6.2 of the CPython interpreter was used along with Sun Java(TM) SE Runtime Environment (build 1.6.0_13-b03) and Jython 2.50.

Table 2. Results of testing various CSP libraries against the Commstime benchmark. In each case the default Channel class is used.

CSP implementation	Process reification	min (μs)	max (μs)	mean (μs)	standard deviation
JCSP	JVM thread	15	29	23.8	4.29
PyCSP	OS process	195.25	394.97	330.34	75.82
PyCSP	Python thread	158.46	311.2	292.2	47.21
PyCSP	Greenlet	24.14	25.37	24.41	0.36
`python-csp`	OS process	67.6	155.97	116.75	35.53
`python-csp`	Python thread	203.05	253.56	225.77	17.51
`jython-csp`	JVM thread	135.05	233	157.8	30.78

The results show that channel operations in `jython-csp` are faster than channel operations between `python-csp` objects when reified as threads, but slower than the thread-based version of `python-csp`. This is a surprising result, as Java threads are better optimised than Python threads (because of the way the Python GIL is implemented) and, as the results for JCSP show, it is possible to implement CSP channels very efficiently in pure Java. The loss of performance is likely to be due to the way in which methods are invoked in Jython. Rather than all compiling Jython code directly to Java bytecode (as was possible when Jython supported the jythonc tool), Jython wraps Python objects in Java at compile time and executes pure Python code in and instance of a Python interpreter. Mixing Python and Java code, as the `jython-csp` library does, can therefore result in poor performance. It may be possible to ameliorate these problems by implemented more of the `jython-csp` library in pure Java code. It is also possible that future versions of Jython will improve the performance of method invocation and/or provide a method of compiling Python code directly to Java bytecode.

The difference between the `python-csp` and PyCSP libraries is also surprising. `python-csp` implements channel synchronisation in a simple manner, with two semaphores protecting each channel, and two reentrant locks to guard against conflicts between multiple readers and/or writers. PyCSP has a very different architecture and synchronisation strategy which may account for the difference in performance. More detailed profiling of the two libraries, together with the completion of the model checking work described in Section (to ensure that `python-csp` is not simply faster because it is somehow incorrect) will form part of our future work.

3.1. Related Work

There are some differences between the implementation of `python-csp` and other realisations of CSP, such as occam-π, JCSP [7] and PyCSP [6]. In particular, any channel object may have multiple readers and writers. There are no separate channel types such as JCSP's `One2OneChannel`. This reflects the simplicity that Python programmers are used to and the PEP20 [1] maxim that ""There should be one—and preferably only one—obvious way to do it.". Also, when a `Channel` object (or variant of such) is instantiated, the instance itself is returned to the caller. In contrast, other systems return a "reader" and "writer" object, often implemented as the read and write method of the underlying channel. This is similar to the implementation of operating system pipes in many libraries, where a reader and writer to the pipe is returned by a library call, rather than an abstraction of the pipe. The authors of these other CSP realisations would argue that their design is less likely to be error prone and that

they are protecting the error-prone programmer from inadvertently reading from a channel that is intended to be a “output” channel to the given process or writing to a channel that is intended to be an “input” to the process. However, `python-csp` comes with strong tool support which may ameliorate some of these potential errors, and the profusion of channel types in some systems may confuse rather than simplify. Similarly, `Alt.select` methods return the value read by a guard rather than the index to the selected guard (as in JCSP) or a reference to the selected guard (PyCSP). The last guard selected is stored in the field `Alt.last_selected` and so is available to users.

Some pragmatic concessions to the purity of `python-csp` have been made. In particular, the three default channel types (`Channel`, `FileChannel` and `NetworkChannel`) have very different performance and failure characteristics and so are implemented separately and conspicuously to the user. The chances of a network socket failing, and the potential causes of such a failure, differ greatly from that of an operating system pipe. Equally, a process which times-out waiting for a channel read will need to wait considerably longer for a read on a `FileChannel` than a `Channel`. These are non-trivial differences in semantics and it seems beneficial to make them explicit.

4. Correctness of Synchronisation in python-csp

To verify the correctness of synchronisation in python-csp, a formal model was built using high level language to specify systems descriptions, called PROMELA (a PROcess MEta LAnguage). This choice is motivated by convenience since a large number of PROMELA models are available in the public domain and some of the features of the SPIN (Simple PROMELA INterpreter) tool environment [12], which interprets PROMELA, greatly facilitate our static analysis. PROMELA is a non-deterministic language, loosely based on Dijkstra’s guarded command language notation and borrowing the notation for I/O operations from Hoare’s CSP language.

The model was divided into the two primary processes: the read and the write. Synchronisation was modeled by semaphores for several readers and several writers in PROMELA. Process type declarations consist of the keyword proctype, followed by the name of the process type, and an argument list. For example, *n* instances of the process type read are defined as follows:

```
active [n] proctype read()
{       do
        :: (rlock) -> rlock = false; /*obtain rlock*/
           atomic{      /*wait(sem)...acquire available*/
                 available > 0 -> available--
           }
           c_r?msg; /*get data from pipe...critical section*/
           taken++; /*release taken*/
           rlock = true; /*release rlock*/
        od;
}
```

The do-loop (terminated by *od*) is an infinite loop. The body of the loop obtains the *rlock* flag to protect from races between multiple readers, then blocks until an item becomes available in the pipe, then gets the item from the pipe (in the critical section), then announces the item has been read, then releases the *rlock* flag. The :: symbol indicates the start of a command sequence block. In a do-loop, a non-deterministic choice will be made among the command sequence blocks. In this case, there is only one to choose from. The write process is declared in a similar style to the read process. The body of the do-loop obtains the *wlock* flag

to protect from races between multiple writers, then places an item in the pipe, then makes the item available for the reader, then blocks until the item has been read, and finally releases the *wlock*.

```
active [n] proctype write()
{       do
        :: (wlock) -> wlock = false; /*obtain wlock*/
           s_c!msg; /*place item in pipe...critical section*/
           available++; /*release available*/
           atomic {
                taken > 0;
                taken-- /*acquire taken*/
           }
           wlock = true; /*release wlock*/
        od;
}
```

The channel algorithm used in this model is defined as:

```
chan s_c = [0] of {mtype}; /*rendezvous channel*/
chan c_r = [0] of {mtype};

active proctype data()
{       mtype m;
        do
        :: s_c?m -> c_r!m
        od
}
```

The first two statements declare s_c and c_r to be channels. These will provide communication to write into or read from the pipe, with no buffering capacity (i.e., communication will be via rendezvous) that carries messages consisting of a byte (*mtype*). The body of the do-loop retrieves the received message and stores it into the local variable *m* on the receiver side. The data is always stored in empty channels and is always retrieved from full channels.

Firstly, the model was run in SPIN random simulation mode. The SPIN simulator enables users to gain early feedback on their system models that helps in the development of the designer's understanding of the design space before they advance in any formal analysis. However, SPIN provides a limited form of support for verification in terms of assertion checking, i.e. the checking of local and global system properties during simulation runs. For example, a process called monitor was devoted to assert that a read process will not be executed if the buffer is empty and the buffer can not be overwritten by write process before the read process is executed.

```
proctype monitor()
{       do
        :: assert(taken <2) ;
        :: assert(available <2);
        od
}
```

Figure 2 shows the Message Sequence Chart (MSC) Panel. Each process is associated with a vertical line where the "start of time" corresponds to the top of the MSC moving down with the vertical distance represents the relative time between different temporal events. The message passing is represented by the relative ordering of arrows between process execution lines.

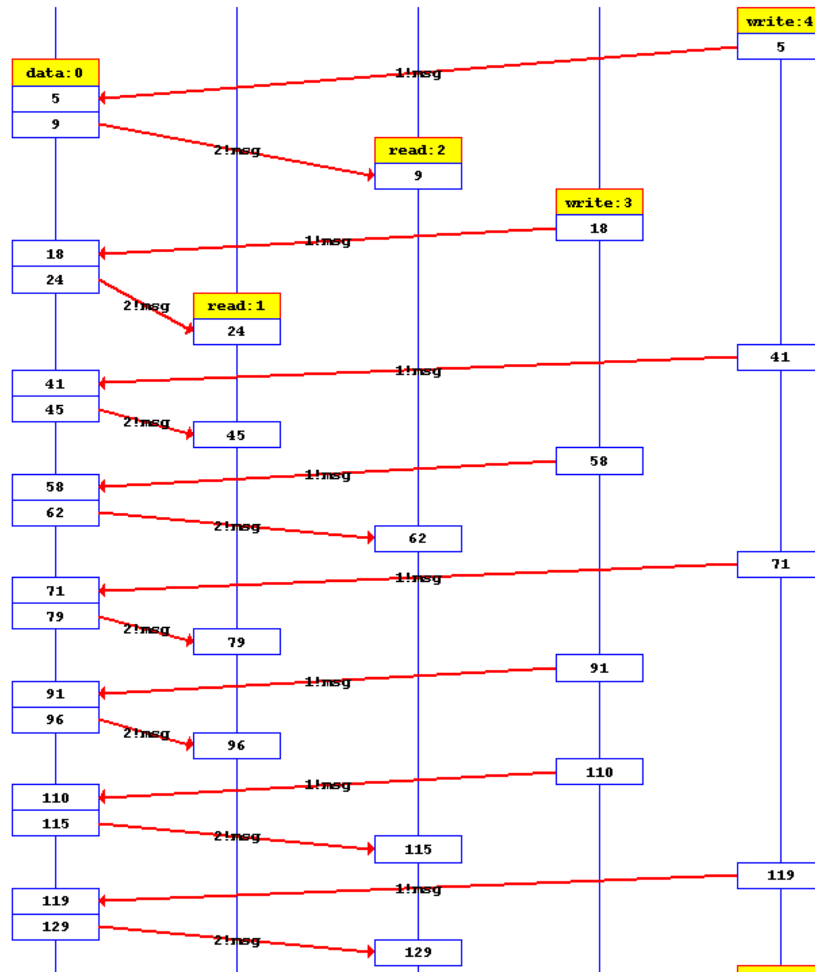

Figure 2. Message sequence chart of SPIN model checker showing two readers and two writers.

SPIN verification mode is used to verify liveness and safety properties like deadlock detection, invariants or code reachability. Verification parameters are set to enable check for "invalid endstates" in the model. The verification output does not show any output referring to "invalid end states" which means that the verification has passed.

5. Conclusions and Future Work

`python-csp` provides a "Pythonic" library for the structuring of concurrent programs in a CSP style. This provides an alternative to the event-driven style which has become prevalent with the increasing popularity of object oriented methods. `python-csp` realises the three fundamental concepts of CSP, processes, synchronous channel communication and non-deterministic choice, for use both explicitly and with appropriate syntactic sugar to provide program texts with much more of the "look and feel" of CSP.

`python-csp` has been realised as a number of distinct realisations. One notable implementation is `jython-csp`, which, as a result of Jython's reliance on the Java Virtual Machine, yields a platform independent implementation.

As an example of a real program, parallelised using `python-csp`, the Mandelbrot generator has been presented. Both a producer-consumer and worker-farmer implementation have been described, and the worker-farmer shows a linear performance relationship with the number of processes used (running on a dual-core computer).

The correctness of channel synchronisation in `python-csp` has been demonstrated using a model checker. Future work will include a study of the correctness of non-deterministic

selection in `python-csp`.

Evaluation of the performance of `python-csp` shows that it performs slightly faster than equivalent implementations of PyCSP (significantly faster for the OS process version).

The motivation for the constriction of `python-csp` was to provide a syntactically natural and semantically robust framework for the design and implementation of large scale, distributed, parallel systems, in particular wireless sensor networks. It is hoped that such systems will one day be grounded in a secure theory of communication and concurrency. `python-csp` has provided such a framework, but is so far limited to a shared memory, single machine implementation. The next stages in this work are to extend the synchronised communications to operate over inter-machine communications links. In some ways, CSP communications, being already modelled on a "channel", are ideal for such a purpose. On the other hand, real communications channels, particularly wireless ones, have quite different characteristics from the instantaneous and reliable CSP channel. Finding efficient means for duplicating the semantics of the CSP channel using real communications remains a challenge for the authors.

Acknowledgements

The authors wish to acknowledge the Nuffield Foundation's support for this research through an Undergraduate Research Bursary (URB/37018), which supported the work of the third author.

References

[1] Tim Peters. PEP 20: the Zen of Python. `http://www.python.org/dev/peps/pep-0020/`, August 2004.

[2] Christian Tismer. Continuations and Stackless Python or "how to change a paradigm of an existing program". In *Proceedings of the 8th International Python Conference*, January 2000.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[4] S.N.I. Mount, R.M. Newman, and E.I. Gaura. A simulation tool for system services in ad-hoc wireless sensor networks. In *Proceedings of NSTI Nanotechnology Conference and Trade Show (Nanotech'05)*, volume 3, pages 423–426, Anaheim, California, USA, May 2005.

[5] S. Mount, R.M. Newman, E. Gaura, and J. Kemp. Sensor: an algorithmic simulator for wireless sensor networks. In *Proceedings of Eurosensors 20*, volume II, pages 400–411, Gothenburg, Sweden, 2006.

[6] John Markus Bjørndalen, Brian Vinter, and Otto J. Anshus. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[7] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[8] Neil C.C. Brown and Peter H. Welch. An introduction to the Kent C++CSP library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 139–156, Amsterdam, The Netherlands, September 2003. IOS Press.

[9] Bernhard H.C. Sputh and Alastair R. Allen. JCSP-Poison: Safe termination of CSP process networks. In Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors, *CPA*, volume 63 of *Concurrent Systems Engineering Series*, pages 71–107. IOS Press, 2005.

[10] Peter H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[11] Peter H. Welch and Fred R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[12] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.