PyCSP Revisited

Brian VINTER [a,1], John Markus BJØRNDALEN [b] and Rune Møllegaard FRIBORG [a]
[a] *Department of Computer Science, University of Copenhagen*
[b] *Department of Computer Science, University of Tromsø*

Abstract. PyCSP was introduced two years ago and has since been used by a number of programmers, especially students. The original motivation behind PyCSP was a conviction that both Python and CSP are tools that are especially well suited for programmers and scientists in other fields than computer science. Working under this premise the original PyCSP was very similar to JCSP and the motivation was simply to provide CSP to the Python community in the JCSP tradition. After two years we have concluded that PyCSP is indeed a usable tool for the target users; however many of them have raised some of the same issues with PyCSP as with JCSP. The many channel types, lack of output guards and external choice wrapped in the select-then-execute mechanism were frequent complaints. In this work we revisit PyCSP and address the issues that have been raised. The result is a much simpler PyCSP with only one channel type, support for output guards, and external choice that is closer to that of occam than JCSP.

Keywords. Python, CSP, PyCSP, alternation, concurrency

Introduction

When PyCSP was introduced in 2007 [1] it was a CSP [2] library in the JCSP [3,4,5] tradition and primarily targeted pedagogical purposes. After having worked with PyCSP for another two years the authors decided to evaluate the experiences made and decide on the future of PyCSP. The outcome of the evaluation had three potential conclusions:

1. PyCSP was a nice exercise but of little or no practical use and the project should be stopped
2. PyCSP is a success as it is and no further work is needed, thus the research should be stopped
3. PyCSP has shown potential but needs more work and/or alternative approaches

Having included PyCSP in the Extreme Multiprogramming class at the University of Copenhagen three years in a row with a combined number of students in excess of 200, we did have a sizable set of inputs on PyCSP. On the upside the students claimed to like PyCSP for a number of reasons:

- It is Python and thus perceived to be easier to work with than most other languages[2]
- The fact that PyCSP channels are type indifferent[3] is convenient when changing the functionality in an application

On the downside, a number of students also had reservations with PyCSP:

[1]Corresponding Author: *Brian Vinter, Department of Computer Science, University of Copenhagen, DK-2100 Copenhagen, Denmark*. Tel.: +45 3532 1421; Fax: +45 3521 1401; E-mail: `vinter@diku.dk`.

[2]This may only be true in the context of this class where the focus is on scientific applications.

[3]The type indifference is easy because Python is dynamically typed.

- The many channel types make code less intuitive and any-to-any was the de-facto choice, though it does not support external choice
- No real parallelism unless functionality is written in C

Going through the final reports for the last exam, we discovered that more than 80% of the students had chosen PyCSP for their solution, second was JCSP, then followed C++CSP. A single report used occam. While some of the success of PyCSP is bound to be due to veneration for a locally developed system, there is little doubt that the students do like PyCSP: Java is the usual language of choice in other classes. It was especially interesting that students with a non-CS background, such as math, physics, nano-science and biology, all chose PyCSP, which indicates that our original intention of making a system for multi-core programming for scientists is within reach.

We decided that option 3, "PyCSP has shown potential but needs more work and/or alternative approaches", was the conclusion of our evaluation and went on to address the input we have gotten from the many users.

The most frequent comment we received was disappointment that true parallelism could not be obtained using pure Python code. This is because Python uses a global interpreter lock, GIL, which means that threads in Python are useful only if a thread calls outside Python or to handle asynchronous events. To address this, the new implementation supports operating system processes in addition to threads. Strictly speaking this could be done with no changes to PyCSP and a new process-based implementation could transparently replace the old one. However, a number of other comments we received addressed the syntax and semantics of PyCSP and we thus decided to revisit the design. The work on the new implementations is presented in another paper [6].

It was also decided to make changes to the PyCSP API. Originally, PyCSP had been inspired by the other CSP libraries – most importantly JCSP – but it was evident that many students found the compact expressions in occam, especially the representation of external choice, attractive. While students easily understood why external choice on output channels is not needed in CSP, they still, rightly, claimed that they would be convenient. Finally, termination through poisoning was easily understood but also claimed to be inconvenient. Thus we decided to change PyCSP in four major ways:

1. There should be only one channel type, any-to-any, and it must support external choice
2. The channels should support both input and output guards for external choice
3. PyCSP should provide a mechanism for joining and leaving a channel with support for automatic poisoning of a network
4. The expressive power in Python should be used to make PyCSP look more like occam where possible

In the following we describe the new PyCSP library based on the above four design criteria. The result is a PyCSP implementation that follows the decisions and, in our own opinion, makes PyCSP programs even more readable and maintainable.

1. The New PyCSP

1.1. Processes

Just as in the original PyCSP, processes are wrapped in a process decorator, i.e. they are not merely implementations of a Process class as in JCSP or C++CSP. The advantage of this approach is partly that processes will be easily recognizable in the source code, and that it gives great flexibility for the PyCSP runtime environment to handle processes in different ways.

The constructor used is `@process`, and a hello world example could look like the following example:

```
@process
def hello_world(msg):
    print "Hello world, this is my message " + msg
```

Usually one or more channel ends will be part of the parameters for a process. Defining a process as above will not instantiate or execute any code: it is simply defined as a process to be used in a network at a later time.

1.2. Process Sets

Once a process is defined, a set of processes may be instantiated and executed using the Parallel or Sequential constructs similar to the old version. However, in order to accommodate variable size networks a process set may now include lists of processes as well as individual processes.

```
Parallel(
    source(),
    [worker() for i in range(10)],
    sink()
)
```

In the above example source, worker and sink have all been defined as processes and the parallel construct will run one source, ten workers and one sink process in parallel and return once all processes have terminated. Naturally the example makes little sense without the use of channels for communication; these will be introduced below. Apart from the support for mixing scalars and vectors of processes, the Parallel and Sequential constructs work as in the previous version and should be intuitive to anybody with any CSP experience.

1.3. Channels

PyCSP originally based much of its design on JCSP, continuing the use of specialized channel types: One2One, One2Any, Any2One and Any2Any. The type names designate how many writer and reader processes were allowed to be attached to the respective channel ends.

The main reason for the specialized channel types was that the implementation of the Alternative construct, which allowed external choice, was based on the JCSP version and placed strict limitations on the use of channels: only one process could safely use an Alternative construct with a given channel end. To safeguard against misuse, only the reading end of channel types that were restricted to one reader could be used as guards in an external choice. Limitations such as these can be cumbersome to work around when designing your CSP application and even more so for newcomers to PyCSP.

1.3.1. New Channel Type

There is only one channel type in the new PyCSP. The channel is similar to the previous Any2Any channel, but with the difference that both input and output channel ends support external choice. The use of external choice is described in section 1.4.

Retrieving channel ends for use in processes has also changed in PyCSP. Previously, a programmer would grab a channel end by calling the `read()` or `write()` method of the channel. This has been replaced with the `channel.reader()` and `channel.writer()` functions which also have a role in channel poisoning described below. As an experiment a shorthand for `channel.reader()` and `channel.writer()` is introduced as `-channel` and `+channel`; whether this more compact notation introduce more confusion than it is worth is left to future observations.

1.3.2. Channel Poison

The concept of poisoning channels with the purpose of shutting down an application was introduced in C++CSP [7] and later investigated in some detail by Bernhard Sputh [8]. A channel is poisoned and all subsequent reads or writes on this channel will throw an exception. This exception can be caught and used as a shut-down procedure or just to shut down that single channel. In the following example we create two processes, `source` and `sink`, and a channel to connect them. The `source` process finally poisons the channel to terminate the network, which will happen since the `sink` process does not catch the exception.

```
@process
def source(chan_out):
    for i in range(10):
        chan_out("Hello world")
    poison(chan_out)

@process
def sink(chan_in):
    while True:
        print chan_in()

chan = Channel()
Parallel(source(chan.writer()), sink(chan.reader()))
```

Since all channels now support multiple readers and writers it is easy to add more readers and writers:

```
Parallel(source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()))
```

or

```
Parallel([source(chan.writer()) for i in range(5)],
         [sink(chan.reader())   for i in range(5)])
```

Both versions produce five source and five sink processes, however the created network will not do what the user may intuitively think it does. One of the sources is bound to finish first and it will then poison the channel, which will terminate the network before all the expected messages have been printed. The problem is extremely common in producer-consumer class applications, and users end up with complex solutions for terminating the network.

To address this we introduce a poison mechanism similar to reference counting. Creating channel ends and retiring from them updates a counter of how many readers or writers we have on a channel, and the leave method may perform automatic poisoning when no readers or no writers are left.

The `reader()` and `writer()` methods automatically join the respective ends of a channel, returning a unique reference to that channel end. A new function, `retire()`, is used to leave a channel end. All subsequent requests to this channel end reference will raise an exception. When all readers or writers have retired a channel, the other end of the channel is also retired. This is similar to how poison is propagated in the previous versions of PyCSP, but with one important difference: with a poisoned channel any reference to that channel will trigger a ChannelPoisonException which is caught in the Process class that wraps all PyCSP processes. The exception handler then poisons all the other channels that were passed to the

process upon initialization. With a retired channel, the ChannelRetireException is thrown and the other channel ends are retired rather than poisoned, implementation wise the two are identical apart from the name of the exception that is raised. This can remove some potential race conditions when terminating networks, as seen in the Monte Carlo Pi example in section 3 and in the example below.

The following code demonstrates how the retire expression can be used instead of the poison expression. The network will now be poisoned by the last source process to finish, rather than the first. This feature hugely simplifies many networks.

```
@process
def source(chan_out):
    for i in range(10):
        chan_out("Hello world")
    retire(chan_out)
```

1.4. External Choice

One of the criticisms that the original PyCSP attracted was the way that external choice was implemented, which had more in common with UNIX socket programming using select than the more compact occam ALT operation. After executing an external choice (Alternative) you are required to read from the selected channel. Failing to do so would break the rules for the choice construct in CSP. Thus we decided to simplify the usage of Alternative by combining select with a custom-defined action on the guard, similar to the occam ALT. Based on this, we introduce a new choice named Alternation.

Alternation has changed significantly from Alternative, partly to make it more like occam, partly to support output guards. A guard set is now represented as a list of Python dictionaries where the keys can be channels from which to read, or two-tuples where the first entry is a channel and the second the value that should be written to that channel. The value of each dictionary entry is a function of type choice which may be executed if the guard becomes true. If the guard is an input guard then the choice function will always have the parameter `__channel_input` available which is the value that was read from the channel. Alternation also supports other guard types, inheriting from a common Guard class. Alternation has two calls:

- Execute – which waits for a guard to complete and then executes the associated choice function, similar to the occam ALT instruction
- Select – which returns a two-tuple: the guard that was chosen by Alternation and, if the guard was an input-guard, the message that was read. This is equivalent to the original Alternative

Note that the execute call in alternation always performs the guard that was chosen, i.e. channel input or output is executed within the alternation, so even the empty choice or a choice function where the results are simply ignored still performs the guarded input or output.

The code that is executed within a guard may be specified in two ways, either as a function that is defined using a choice decorator, similar to processes, or as a string containing code to be executed. The latter is easy to use but becomes quite slow since runtime compilation is required. A choice function is defined as follows:

```
@choice
def action(__channel_input=None):
    print __channel_input
```

It is not possible to change the name of `__channel_input` in a choice function since it is passed as a keyword argument when it is the result of a selected input guard. Once the choice is

defined, a process may perform an alternation on a set of channel ends. In the following example the same guarded code, action, is called independently of which channel becomes ready. This is an option but naturally not a requirement.

```
@process
def par_reader(cin1, cin2, cin3, cin4):
    Alternation([
        { cin1:action() },
        { cin2:action() },
        { cin3:action() },
        { cin4:action() }
    ]).execute()
```

An action might alternatively be passed as a string. This string is then evaluated with a copy of the current namespace. All mutable types can be updated from the evaluation of this string. In Python, the `list`, `dict` and `set` types are built-in mutable types.

```
@process
def counter(cin0, cin1):
    try:
        cnt = [0, 0] # use mutable type
        while True:
            Alternation([
                { cin0: 'cnt[0] += 1' },
                { cin1: 'cnt[1] += 1' }
            ]).execute()
    except ChannelPoisonException:
        print 'Counted:', cnt
```

Guards are prioritized in the order they occur in the guard list, while guards in a dictionary are unordered. This gives us the option to model both an ordinary external choice and a prioritized external choice. It is important to note that priority only makes real sense in the scenario where more than one guard is ready when the alternation is entered, which guard is woken first if no guards are immediately available may be down to race-condition or the priority order of another guard statement.

Ordinary external choice is obtained by a list with just one dictionary holding guards. The entries in the dictionary are then treated in an non-prioritized way:

```
[{
    cin1:action(),
    cin2:action(),
    cin3:action()
}]
```

On the other hand, prioritized external choice is obtained by providing a list of dictionaries with guards. These are then prioritized in the way the dictionaries appear in the list.

```
[
    { cin1:action() },
    { cin2:action() },
    { cin3:action() }
]
```

It is fully possible to mix the two models, i.e. a prioritized list with dictionaries of non-prioritized guards. This option should only be used for special purposes.

PyCSP provides four built-in guard types to use with external choice. The first three of them are well known to the CSP community:

- Channel input
- Timeout - A counter relative to current time, when it expires, it will become true and allow the alternation to complete
- Skip - Always true, and often used to define a default alternative
- Channel output

The fourth is new in PyCSP, although thoroughly discussed throughout the years and previously seen in Communicating Java Threads [9]. It is well understood by most programmers that use process algebra that output guards are not needed from a CSP point of view, and one may with relative ease construct equivalences for any type of output guard using only input guards. However, output guards are convenient for the user of PyCSP and equivalences are hard to construct for users that are not professional programmers, thus we provide the output guard as a primitive in PyCSP. All guard types supported can be interrupted by channel poisoning or retiring. PyCSP channels may be guarded in both ends, i.e. an output guard can be matched by an input guard.

The following code example shows how non-blocking writes and input with timeouts can be modelled using the new Alternation construct:

```
# Non-blocking write
Alternation([
    { (cout, datablock): None },  #Try to write to a channel
    { Skip(): "print 'skipped!'" }  #Skip the alternation
]).execute()

# Input with timeout
Alternation([
    { cin: "print __channel_input"},
    { Timeout(seconds=1): "print 'timeout!'" }
]).execute()
```

2. Implementation

This section introduces only highlights of the implementation. An in depth description of the implementation details of PyCSP may be found in [6].

The only implementation detail that is non-trivial is the support for output guards and channels with multiple processes at either end. The implementation is quite complex and uses more than a hundred lines of Python code. The overall design is based on each alternation being represented by a request structure, called a handle in the pseudocode below, that includes a lock which ensures mutual exclusion. When a new alternation is activated it will traverse the guards in the choice list by priority, and for each guard it will look for a waiting handle that matches the handle for the alternation, i.e. a read matches a write and vice versa. If no match is found, the handle is added to the set of waiting handles for that channel. Please note that the pseudocode is heavily simplified and the actual implementation relies on global ordering of events to avoid livelocks; for details refer to [6].

```
handle = new_request_handle()
for guard in choice:
    lock(guard.channel)
    if handle match registered_handle in guard.channel:
        perform communication
```

```
            make_active(handle, registered_handle)
        else:
            guard_channel.registered_handle.add(handle)
        unlock(guard.channel)
    waitfor active(handle)
```

This is the procedure for every channel communication possible. Whenever a match is tried, two locks are required: one owned by the reading end and one owned by the writing end. In the case of alternation, this lock is shared between all guards to ensure the integrity of the alternation. A diamond design where every process alternates on an input and an output end could look similar to the example code below.

```
@process
def P(id, c1, c2):
    while True:
        Alternation([{(c1, True): None, c2: None}]).select()

c = [Channel(str(i)) for i in range(4)]

Parallel(
    P(1,c[0].writer(), c[1].reader()),
    P(2,c[1].writer(), c[2].reader()),
    P(3,c[2].writer(), c[3].reader()),
    P(4,c[3].writer(), c[0].reader())
)
```

Without acquiring the locks in perfect order, this code eventually results in a deadlock. Two processes have both acquired one of the alternation owned locks and are waiting to acquire the other in opposite order. To acquire the locks in order, we always acquire the lock with the lowest memory address first. This ensures the same lock order for all processes.

We are synchronizing input and output guards without the *Oracle* process used in JCSP [10]. The *Oracle* process was introduced in JCSP to handle external choice on barriers and output guards. The new PyCSP views all communication requests as an offer and all offers are protected by individual locks. One lock per offer eliminates the need for an *Oracle* process, because it is guaranteed that an offer is only matched while it is active. The purpose of the *Oracle* process is to ensure that an offer is still active, when it is matched.

3. Examples

Some of the changes in PyCSP refer to either performance and implementation, or pure syntactical presentation of the concepts. In the following, we show two examples that motivate the three semantic changes that have been introduced: retire as an alternative to channel poisoning, output-guards, and support for alternation with channels that have multiple readers and/or writers. The purpose of the examples is to demonstrate why PyCSP becomes easier to use after the introduced changes.

3.1. Monte Carlo Pi

In the original PyCSP we did not have the retire feature, which meant that most producer-consumer programs tended to look like the example in figure 1.

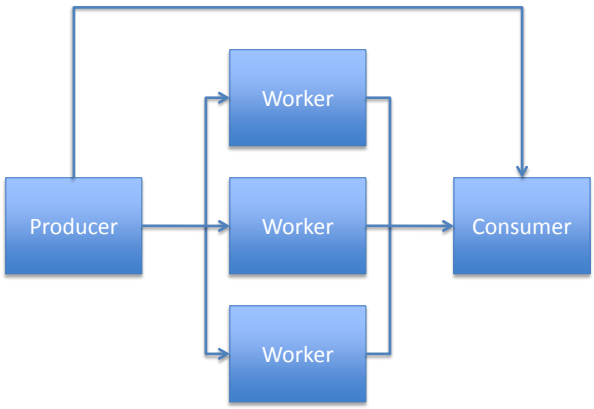

Figure 1. Producer-consumer program forwarding termination criteria between producer and consumer.

```
from pycsp import *
from random import random

@process
def producer(term_out, job_out, bagsize, bags):
    term_out(bags)
    for i in range(bags): job_out(bagsize)
    poison(job_out)

@process
def worker(job_in, result_out):
    try:
        while True:
            cnt = job_in()              #Get task
            sum = reduce(lambda x, y: x+(random()**2+random()**2<1.0),
                         range(cnt))
            result_out((4.0*sum)/cnt)   #Forward result
    except ChannelPoisonException:
        pass                            #When done, _don't_ forward poison
@process
def consumer(term_in, result_in):
    cnt = term_in()           #Get number of results
    sum = 0
    for i in range(cnt):
        sum += result_in()    #Get result
    print sum/cnt             #We are done - print result

jobs = Channel()
results = Channel()
term = Channel()

Parallel(producer(term.writer(), jobs.writer(), 1000, 10000),
         [worker(jobs.reader(), results.writer()) for i in range(10)],
         consumer(term.reader(), results.reader()))
```

Listing 1. Implementation of producer-consumer program forwarding with an explicit termination channel between producer and consumer.

A simple Monte Carlo simulation of the design in figure 1 is implemented in listing 1. This approach is simple to implement but suffers from two complexities: first of all, the termination criterion (number of bags) must be sent from the producer to the consumer, bypassing the workers. Secondly, the workers must explicitly avoid forwarding the channel poison in

the network as the consumer will otherwise die before all results are received and processed. The complexity of the solution grows even further if the setup has multiple producers or consumers.

With the new retire operation, termination of the network can be handled in a more straightforward way. When the producer retires the `job_out` channel end, the channel is not poisoned, but the retire operation is forwarded to the other end of the channel in a similar way to poisoning. The main difference is that when the channel is retired and `job_in()` terminates the worker process, the workers channels are retired rather than poisoned. This will delay termination propagation along those channels until all workers have retired, and the consumer will not be prematurely terminated.

Figure 2 shows the new network, which no longer needs to forward a termination criterion between the producer and consumer process. Multiple producers can also be plugged into the network without changing more than the parallel construct. Note that the consumer in listing 2 catches a ChannelRetireException, which allows it to terminate cleanly and print out the results before terminating.

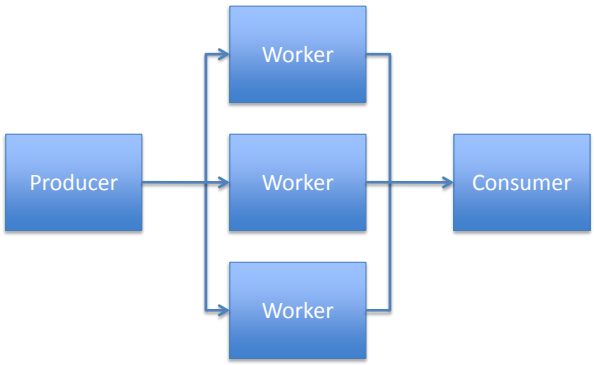

Figure 2. Producer-consumer program using retire, avoiding termination criteria forwarding between producer and consumer.

```
from pycsp import *
from random import random

@process
def producer(job_out, bagsize, bags):
    for i in range(bags): job_out(bagsize)
    retire(job_out)

@process
def worker(job_in, result_out):
    while True:
        cnt = job_in()          #Get task
        sum = reduce(lambda x, y: x+(random()**2+random()**2<1.0),
                     range(cnt))
        result_out((cnt, sum))  #Forward result

@process
def consumer(result_in):
    cnt, sum = 0, 0
    try:
        while True:
            c, s = result_in() #Get result
            cnt, sum = cnt+c, sum+s
    except ChannelRetireException:
        print 4.0*sum/cnt      #We are done - print result
```

```
jobs = Channel()
results = Channel()

Parallel(producer(jobs.writer(), 1000, 10000),
         [worker(jobs.reader(), results.writer()) for i in range(10)],
         consumer(results.reader()))
```

Listing 2. Producer-consumer using retire, avoiding termination criteria forwarding between producer and consumer.

3.2. Branch-and-bound

Branch-and-bound algorithms in CSP are easy [11] but associated with some complex decisions with regards to how and when to update the bound variable. The challenge in the bound update is to balance the communication and work: if the bound variable is updated too rarely the parallel work will perform more work than necessary. If the bound is updated too often, it will result in too frequent communication. Basically, three approaches exist:

1. Update the bound only when a job is finished. Submitting a bound equals requesting a job
2. Update the bound as soon as you find it. A special bound value identifies a job request
3. Update the bound as soon as you find it. Jobs are requested independently

The first approach is simple and a common choice; however, the infrequent update of the bound results in slower overall execution. The second approach is complex and requires parsing of the input to determine if an incoming message requires an outgoing job. The third is easy but requires output guards.

To keep this section from growing too large we only present the code required for receiving bound variables and passing on jobs.

Solution 1 is the trivial case, where the master sends a new job back when receiving a result from a worker. We don't need an alternation in this case since all channels are any-to-any. When there are no more jobs to be executed, the master retires the job channel which will terminate workers trying to read from it. The master continues to receive results until all workers have retired which will retire the result channel. This will in turn throw a ChannelRetireException in the master, which can be caught to print out the final result.

```
bound = 10e10
while jobs:
    next = jobs.pop()
    bid = results_in()
    bound = best(bid, bound) #Best is an optimization specific function
    jobs_out((next, bound))

#Without retire the code becomes even more complex
retire(jobs_out)

try:
    while True:
        bound = best(results_in(), bound)
except ChannelRetireException:
    print bound
```

Listing 3. Solution 1.

Solution 2 allows workers to submit new bound variables before the job is finished by sending an *update* message over the request channel. This allows the bound variable to be updated faster and thus potentially reduces the total work that must be done. The solution is almost identical to solution 1, except that messages from workers are parsed to know if the message is an update. Updates should not trigger a blocking write of a new job to the jobs channel. Termination is identical to solution 1.

```
bound = 10e10
while jobs:
    next = jobs.pop()
    request, bid = results_in()
    if request == 'Update':      #Update means don't send new job
        bound = best(bid, bound) #Best is an optimization specific function
    else:
        jobs_out((next, bound))

#Without retire the code becomes even more complex
retire(jobs_out)

try:
    while True:
        bound = best(results_in()[1], bound)
except ChannelRetireException:
    print bound
```

Listing 4. Solution 2.

Solution 3 uses output guards to eliminate parsing of the incoming messages. Instead, the alternation accepts either an incoming result or an outgoing job to a worker. Once there are no more jobs, the solution terminates like the other solutions.

It should be noted that this design, which is as simple as solution 1 and as efficient as solution 2, also provides simpler initialization of the workers since they are not required to submit a bogus result to trigger the delivery of the first job.

```
#A Python limitation requires a mutable type here
my_locals = {
   'bound': 10e10,
   'next' : jobs.pop() #We require at least two jobs to start with!!!
}
while jobs:
    Alternation([{
        results_in:
          "my_locals['bound'] = best(__channel_input, my_locals['bound'])",
        (jobs_out, (next, my_locals['bound'])) :
          "my_locals['next'] = jobs.pop()"
    }]).execute()

#Without retire the code becomes even more complex
retire(jobs_out)

try:
    while True:
        my_locals['bound'] = best(results_in(), my_locals['bound'])
except ChannelRetireException:
    print bound
```

Listing 5. Solution 3.

If one wishes the workers to update their knowledge of the bounds, this may easily be done in solution 3 by adding a dedicated channel for propagating the bound and letting the workers do a blocking input from that channel as frequently as desired. The server then adds another output guard that at any time is ready to write the best known bound.

4. Future Work

The new version of PyCSP provides a very convenient means of writing concurrent applications for non-computer scientists, allowing them to use CSP for parallel and concurrent programming. Initial response on the new version has been quite positive and we thus plan to continue the work.

An extension to alternation in JCSP is the concept of fairSelect, which can be used to avoid starvation. This would be of interest also to PyCSP users, probably as default with the truly random alternation becoming an option for special purposes.

Network construction is still fairly complex in PyCSP, and the only improvement that the new version offers is the option of mixing single processes and lists of processes in one Parallel constructor. We are working on a library of network constructors that will allow users to easily specify networks of processes in rings, meshes, fully interconnected and other common process-oriented design patterns.

The previous version of PyCSP was extended with a module that allowed processes to be executed on Grid when channel communication can be represented as a synchronous event, i.e. input;execute;output, the Grid enabled processes cannot support all channel communications, i.e. alternation or patterns as input;execute;input;execute;output cannot be used but classic, client-server patterns fits will with Grid execution. This feature is desired for very demanding jobs and would be relevant to reintroduce in the new version of PyCSP.

While the type indifference of channels in PyCSP is highly praised by students there are scenarios where type matching is equally attractive. Future plans include adding support for type-checking channels.

5. Conclusions

The original PyCSP borrowed heavily from JCSP to get semantics and functionality correct while still attempting to make the solution native to Python. It was quite well received, especially amongst students and scientists who often find Python a productive programming environment. After exposing more than 200 students to PyCSP, we did however receive some negative feedback. One of the central complaints was about the many channel types and especially the hardship of changing between them in an existing application. Another frequent complaint was the lack of support for output guards and channels with multiple readers and/or writers in alternation. In addition to the feedback from the users, the authors identified two shortcomings in the original version of PyCSP: first, students frequently demonstrated race-conditions when terminating a network by use of poisoning, and second, it is desired to make PyCSP look more like occam.

The complaints and identified shortcomings resulted in an evaluation that confirmed the need for the following major changes to PyCSP. All channels are now any-to-any which greatly simplifies design changes since a user may add more readers or writers to a channel that previously had only one. Since external choice is central to CSP, these any-to-any channels are naturally supported in the alternation implementation of external choice.

PyCSP external choice now supports output guards in addition to input guards. This works with multiple readers and writers on a channel. The use of output guards is a heavily debated issue in CSP as they are clearly not needed nor trivial to implement. However, it is

evident that the users of PyCSP find output guards a very convenient feature and considerable work has been put into supporting output guards in the alternation implementation in PyCSP.

External choice has also been modified to more closely mimic occam so that a guard and the associated code can be expressed in one statement. This brings PyCSP much closer to conventional CSP than the previous model where a ready guard was first identified and then read from.

In order to reduce the risk of race-conditions when using poison to terminate a CSP network, this version of PyCSP introduces the concept of retirement from a channel. When all processes on one end of a channel retire their channel ends, the channel becomes retired. The effect is that the propagation of the retire signal is activated upon the termination of the last process at a given channel end rather than the first as with the poison operation.

Overall, the changes to PyCSP are well integrated and we believe that using PyCSP is now easier for the unsophisticated users than with the previous version. The newest version may be found as PyCSP under Google-code [12].

References

[1] John Markus Bjørndalen, Brian Vinter, and Otto Anshus. PyCSP - Communicating Sequential Processes for Python. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, July 2007.

[2] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM, 21(8):666-677*, pages 666–677, August 1978.

[3] Communicating Sequential Processes for Java. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[4] Jim Moores. Native JCSP: the CSP-for-Java library with a Low-Overhead CPS Kernel. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering*, pages 263–273. WoTUG, IOS Press (Amsterdam), September 2000.

[5] 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.

[6] Rune Møllegaard Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In *Communicating Process Architectures 2009*. WoTUG, IOS Press.

[7] Neil C.C. Brown and Peter H. Welch. An introduction to the Kent C++CSP Library. *Communicating Process Architectures 2003*, September 2003.

[8] Bernhard H.C. Sputh and Alastair R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. *Communicating Process Architectures 2005*, September 2005.

[9] Gerald H. Hilderink, Jan F. Broenink, Wiek Vervoort, and Andrè W. P. Bakkers. Communicating Java Threads. In Andrè W. P. Bakkers, editor, *Proceedings of WoTUG-20: Parallel Programming and Java*, pages 48–76, March 1997.

[10] Peter H. Welch, Neil C.C. Brown, Jim Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In Steve Schneider, Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 349–370, Amsterdam, The Netherlands, July 2007. WoTUG, IOS Press.

[11] Peter H. Welch and Brian Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 203–222, September 2002.

[12] PyCSP distribution. `http://code.google.com/p/pycsp`.