**Document Number:** N4464
**Date:** 2015-04-10
**Project:** SG1 - Concurrency
**Reply to:** Cleiton Santoia Silva
<cleitonsantoia@gmail.com>

# Pi-calculus syntax for C++ executors

# Contents

# 1 Intro [intro]

## 1.1 Overview [intro.overview]

My humble attempt to use the expressiveness of Pi-Calculus and static type checking of C++11.
Reference implementation in https://github.com/cleitonsantoia/concurrency
Please learn Pi-Calculus first, than the next line will start to make sense

```
P = a(x) | a[x] | Q * R | (Q | R) | !Q | v{x} | If(pred).then(Q).else(R) | 0
```

Getting started with C++ version of Pi calculus syntax:

Table 1 — Pi-Calculus concepts and s

| | |
|---|---|
| receive x from a | `P = a(x)` |
| send x to a | `P = a[x]` |
| sequence | `P = Q * R, first Q then R` |
| parallel | `P = Q | R, both Q and R in separated threads` |
| replicate | `P = !Q, do Q forever` |
| local | `P = vx, make a local x` |
| decision | `P = If(pred).then(Q).else(R), this is a replacement to Q+R` |
| define an Input channel | `IChannel<>` |
| define an Output channel | `OChannel<>` |
| define a synchronous channel | `Sync<_In, _Out>` |
| define a Process | `Process P` |

So it's possible to define in plain C++ the following statements:

```
Queue<int> queue;
Var<int> p;
Var<int> q;
IChannel<int> producer = [](){ static int index; return ++index; }
OChannel<int> consumer = [](int x){ std::cout << x << std::endl; }
```

Then now those operator() and operator[] are defined for producer and consumer:

```
producer(e)    // get a produced elements into e (forever)
consumer[e]    // send e to consumer
queue[e]       // push e into queue
queue(e)       // pop e from queue
```

The program must have one process to receive the item from `producer` and queue it on `queue`, and another process that get one item from `queue` and send it to the `consumer`.

```
P1 = producer(p) * queue[p];
P2 = queue(c) * consumer[c];
```

Now I want both processes in parallel for ever.

```
P = !P1 | !P2;
```

## 1.2 Full Sample [intro.sample]

The previous sample with a full compilable source code in C++

```
#include <iostream>
#include <thread>

#include "queue.h"
#include "pi.h"

using namespace pi;
using namespace util;

int main() {
        Queue<int> queue(100);c
        Var<int> p;
        Var<int> c;
        IChannel<int> producer([](){
                        static int index;
                        std::cout << "Gen " << ++index << std::endl;
                        return index; } );
        OChannel<int> consumer([](int x){std::cout << "Con " << x << std::endl; });

        // nothing will run, here just preparing
        Process P1 = producer(p) * queue[p];
        Process P2 = queue(c) * consumer[c];
        Process P = !P2 | !P1;

        // now, goes
        std::thread thread(P);
        thread.join();

        return 0;
}
```

# 2 Impact and Motivation [impact]

## 2.1 Impact [impact.impact]

1. This proposal is presenting a way of adapt the structures from some current C++17 papers on concurrency, for instance [Mys14, N4143], [Hal14, N3557] and [GLSM13, N3558] into Pi-Calculus syntax. Some structures from this proposal like parallel and sequence operator can be easily replaced by parallel and sequence executors.

2. The major concern is about the Pi-Calculus syntax itself and not direct comparing the other proposals nor change any of the semantics nor structures of them.

3. The reference implementation presented is not part of the proposal, it's just didactic way of letting developers to experiment pi-calculus.

## 2.2 Motivation [impact.motivation]

1. Use a formal foundation for concurrency.

2. Allow a terser syntax for concurrent code.

# 3 Design [design]

## 3.1 Channel [design.channel]

Is highly recommended that you get some notion of The [Mil99, Pi-calculus] notation and concepts before dig in this document. This [Win02, FAQ] is also recommended.

## 3.2 Var [design.var]

The most intricate structure of the proposal is this one, it's implemented with a `shared_ptr` to a `shared_ptr` to a type, this is needed to:

— Allow communication between processes when the lifetime of the variable declaration is ended;

— Allow a local variable be created at any point in the process, and from that point until the end of process definition, allow communication between processes when the lifetime of the variable declaration is ended;

— Allow pi-calculus polymorphism style, when you send a channel thru a channel and then assign it to a variable, and finally use a variable as a channel;

You can safely use vars in processes and drop their original scope, they still alive when referenced from the process.

```
class QueuedObj {
 Queue<int> queue_;
public:
 Process build() {
  Var<int> x;
  Process P = queue_(x); // ok, this will keep x live
  return P;
 }
}
```

You can use vars as channels.

```
Var<int> v;
Var<int> w;
Queue<int> queue;
Process P = w[10] * v[w] * queue[v] * v(w);
```

— The process P sets w = 10, then sets v = w, then send v to queue, finally set w = v

— `v[x]` sends x to v, same as attribution of value x to v

— `v(x)` sends v to x, same as attribution of value v to x

You can pass channels thru channels, but you must be REALLY careful with this, it's necessary for polymorphic pi-calculus. The rules are the same for other POD types, so when you make a var-channel local, ‘vx‘ apply same rules of any POD (make a local copy). Normally it's not what you want, you must implement a channel handler class that shares the true channel. I think, the best way to deal with this is creating a `Var<T&>` or `Var<std::reference_wrapper>` specialization ( some day I'll do this ), but for now:

```
Var<Event> event;
Queue<Event> event_queue;
Var<IChannel<Event>> handler;
```

```
Queue<IChannel<Event>> queue_handler;
// receive a handler, a event and send the event to it;
Process P = queue_handler(handler) * event_queue(event) * handler[event];
```

## 3.3 Local [design.local]

When you are defining a process, pi-calculus has an operator called "renaming" used to change and the introduce new "local" names to variables and channels to the rest of the process. I did not introduced this syntax because the C++ rhs does not allow us to introduce a new name. So, instead it was implemented a way to just make the variable local. It works only with vars, not with channels, it means that the process maintains alive the vars after the declarations goes out of scope but not the Channels;

```
// an ill formed sample
Process build() {
 Queue<int> q;
 Var<int> x;
 Process P = v{x} * q(x);  // problem: reference q local
 return P;
}

// a Ok sample
// This is ok as far as queue q does not be destroyed during execution of P.
Process build(Queue<int>& q) {
 Var<int> x;
 Process P = v{x} * q(x); // ok P maintains a copy of x and q is not local
 return P;
}
```

The introductory sample will also works with only one var;

```
Var<int> p;
Var<int> c;

Process P1 = producer(p) * queue[p];
Process P2 = queue(c) * consumer[c];
```

Can be modified to use only one var declaration, but two different instances

```
Var<int> e;
Process P1 = v{e} * producer(e) * queue[e];
Process P2 = v{e} * queue(e) * consumer[e];
```

In `P1`, `producer` and `queue` shares the same `e`; In `P2`, `queue` and `consumer` shares the same `e`;
However the `e` in `P1` is different than `e` in `P2`.

## 3.4 Process [design.process]

The process is a wrapper for a `std::function` and a placeholder to help aggregate expressions with `|`, `*` and `!` operators.

You can create ‘Process‘ implicitly with any thing that ‘std::function<void()>‘ recognizes, such as lambdas of plain functions or function-objects.

```
Process P = a(x) * [](){ do_wathever() };  // parameter-less lambdas...
Process Q ([](){ do_wathever() });         // ...may appear as a Process
```

Lambdas that returns values may be used as input channels, any thing compatible with `std::function<T()>` can be used.

```
IChannel<int> input( [](int x){} );
```

Parameterized lambdas may be used as output channels, any thing compatible with `std::function<void(T)>` can be used;

```
OChannel<int> output( [](){ return int(1); } );
```

You can wait for a `std::future` in some lambda.

```
std::future<int> fut;
IChannel<int> rec([fut&](){ return fut.get(); })
```

### 3.5 Assync.Then [design.assync.then]

That's symply the sequence operator `P * Q`

```
Process P = [](){ do_something(); };
Process Q = [](){ do_after_something(); }
Process U = P * Q;   // P then Q
```

You can pass a parameter from P to Q

```
Var v<int> v;
Process P = [](){ return do_something(); };
Process Q = [](int x){ do_after_something(x); }
Process U = P(v) * Q[v];   // receive v via P then send to Q
```

### 3.6 Terse notation [design.terse]

You can create the process from the introduction example with a more direct syntax.

```
IChannel<int> producer = [](){ static int index; return ++index; }
OChannel<int> consumer = [](int x){ std::cout << x << std::endl; }
Process P1 = producer(p) * queue[p];
Process P2 = queue(c) * consumer[c];
thread t(Process(!P1 | !P2));
t.join();
```

Or even more

```
std::thread(
  Process( !(IChannel([](){ static int index; return ++index; })(p) * queue[p]) |
    !(queue(c) * OChannel([](int x){ std::cout << x << std::endl; })[c]) )
).join();
```

# 4 Reference implementation [spec]

## 4.1 Process [spec.Process]

```
class Process {
public:
    Process();
    Process(const int i); // so we can use 0 syntax :)
    Process(const Process& p);
    Process& operator=(const Process& p);
    Process(Process&& p);

    template<typename _Invokable>
    Process(const _Invokable& invokable);

    Process(const std::function<void()>& invokable);

    template<typename _Invokable, typename _Object>
    Process(const _Invokable& invokable, _Object &obj);

    void operator()();
    Process operator!();
};
```

## 4.2 Replication [spec.repl]

```
class Replication {
public:
    Replication(const std::function<void()>& invokable);
    void operator()();
};
```

## 4.3 Variable [spec.var]

```
template<typename _Tp>
class Var {
public:
 Var();
 Var(const Var& var);
 Var(const _Tp& val);

 _Tp& operator*();
 const _Tp& operator*() const ;

 _Tp* operator->()
 const _Tp* operator->() const;

 _Tp& operator=(const _Tp& val);

 template<class _U>
 Var<_Tp>& operator=(const Var<_U>& val);
```

```
  template<typename _U>
  Process operator()(const Var<_U>& var);

  template<typename _U>
  Process operator()(const _U& val);

  template<typename _U>
  Process operator[](Var<_U>& var);

  template<typename _U>
  Process operator[](Var<_U>& var) const;

  template<typename _U>
  Process operator[](_U& val);

  template<typename _U>
  Process operator[](_U& val) const;
};
```

### 4.4 Local operator [spec.local]

```
class v {
public:
    v(const std::initializer_list<VarWrapperRef>& vars);
    void operator()();
};
```

### 4.5 Stop [spec.stop]

```
class Stop {
public:
    void operator()();
};
```

### 4.6 Sequence [spec.sequence]

```
class Sequence {
public:
    Sequence();
    Sequence(const std::initializer_list<Process>& p);
    Sequence& operator*(const Process& p);
    void operator()();
};
```

### 4.7 Parallel [spec.parallel]

```
class Parallel {
public:
    Parallel( const std::initializer_list<Process>& p );
    Parallel( Process p, int num );
    Parallel& operator|(const Process& p);
    Parallel& operator|(const Parallel& p);
    void operator()();
};
```

### 4.8 If [spec.if]

```
class If {
    class ElseT {
    public:
        ElseT(const std::function<bool()>& pred, const Process& then, const Process& else_process);
        void operator()();
    };

    class ThenT {
    public:
        ThenT(const std::function<bool()>& pred, const Process& then);
        ElseT Else(const Process& p);
        void operator()();
    };

public:
    If(const std::function<bool()>& pred);
    ThenT Then(const Process& p);
};
```

### 4.9 IChannel [spec.IChannel]

```
template<typename _Tp>
class IChannel {
public:
 IChannel(const std::function<_Tp()>& invokable);

 template<typename _U>
 Process operator()(Var<_U>& var);
};
```

### 4.10 OChannal [spec.OChannel]

```
template<typename _Tp>
class OChannel {
public:
 OChannel(const std::function<void(const _Tp&)>& invokable) : invokable_(invokable) {}

 template<typename _U>
 Process operator[](Var<_U>& var);
};
```

### 4.11 Synchronous Channel [spec.Sync]

```
template<typename _In, typename _Out = _In>
class Sync {
public:
    template<typename _Invokable>
    Sync(const _Invokable& invokable);

    Sync();

    pi::Process operator()(pi::Var<_In>& var);
    pi::Process operator[](const pi::Var<_Out>& var);
};
```

### 4.12 Some operator overloads [spec.operator]

```
Sequence operator*(const pi::v& p1, const Process& p2);
Sequence operator*(const Process& p1, const pi::v& p2);
Sequence operator*(const Process& p1, const Process& p2);
Parallel operator|(const Process& p1, const Process& p2);
Parallel operator^(const Process& p1, int num);
```

Remeber the names of the classes of the specification are just didactic, the operator syntax and terse notation are the most important.

# Bibliography

[GLSM13] Niklas Gustafsson, Artur Laksberg, Herb Sutter, and Sana Mithani. N3558 a standardized representation of asynchronous operations. Technical report, Programming Language C++, 2013.

[Hal14] Pablo Halpern. N3557 considering a fork-join parallelism library. Technical report, Intel Corp, 2014.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Paperback, 1999.

[Mys14] Chris Mysen. N4143 executors and schedulers, revision 4. Technical report, Programming Language C++, 2014.

[Win02] Jeannette M. Wing. Faq on pi-calculus. Technical report, Professor of Computer Science, Carnegie Mellon University, 2002.