ParTypes in Action

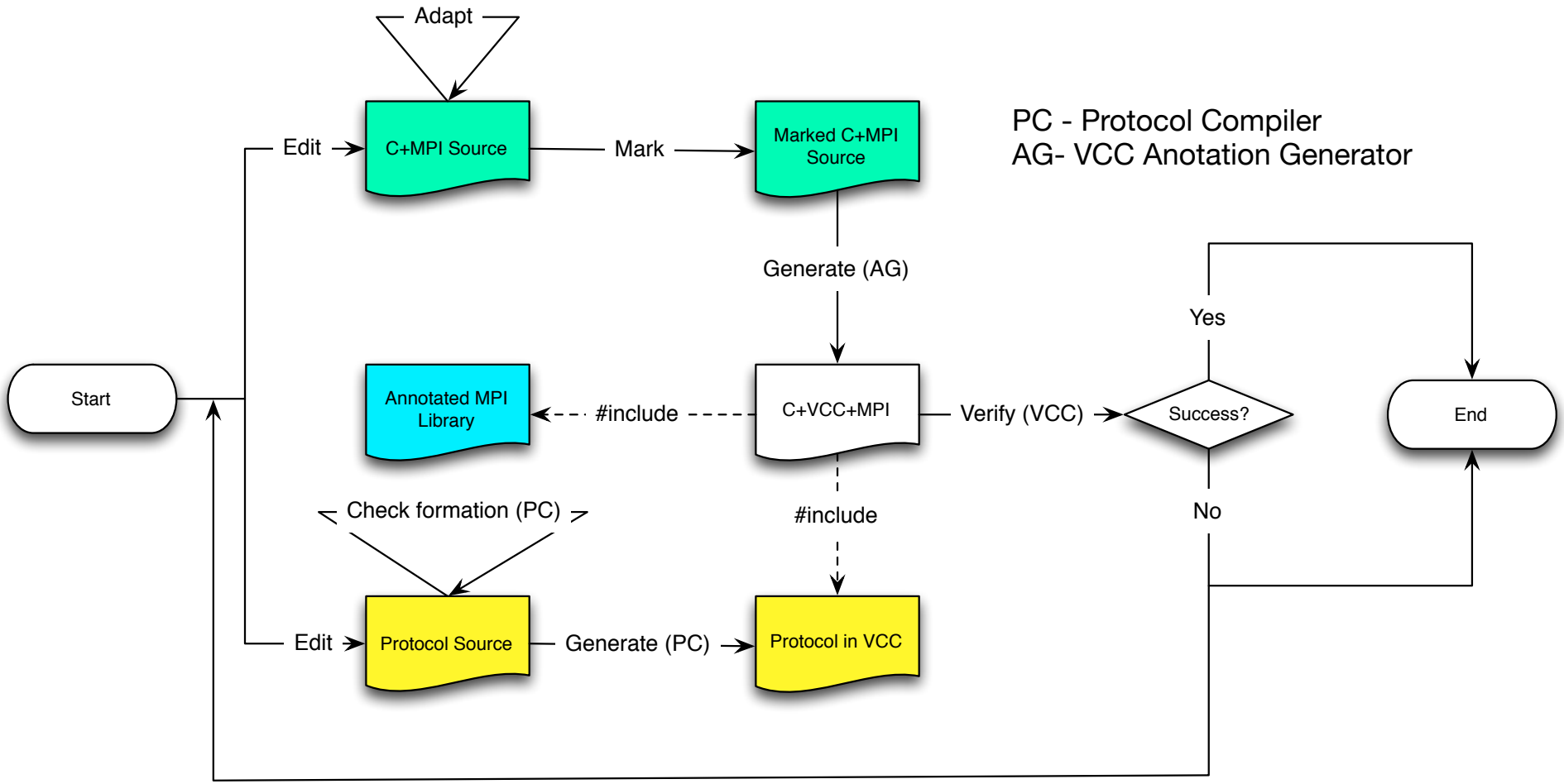
Joint work with V. Vasconcelos, H. Lopez, C. Santos, E. Marques, N. Yoshida, N. Ng

Betty meeting, March 18th, Valletta

Goal

Use (a variant of) MPST to verify that parallel C programs that use MPI (C+MPI) follow a specified protocol

Our approach



Crash course on MPI

- **C+MPI** adhere to the **Single Program Multiple Data** paradigm
  - The same program is deployed in every node
  - Distinct node behaviour is achieved by branching
- There are **hundreds of MPI primitives** for communication and synchronisation
  - This people care for performance
  - A broadcast is not a derived construct
    - Sending a message to every node is not the same as broadcasting a message to all nodes

MPI primitives we cover

- Start and stop MPI
  - **MPI_Init**
    - We start verifying the protocol
  - **MPI_Finalize**
    - We check that the protocol came to an end
- Obtain information about the environment
  - **MPI_Comm_rank**
    - Obtains the process id
  - **MPI_Comm_size**
    - Gives the number of participants in the run

MPI communication primitives

- Point-to-point communication
  - **MPI_Send** and **MPI_Recv** for direct communication between participants
- Collective communication
  - **MPI_Bcast**
  - **MPI_Scatter** and **MPI_Gatter**
  - **MPI_Reduce**
  - **MPI_All_reduce, MPI_All_gather**,…

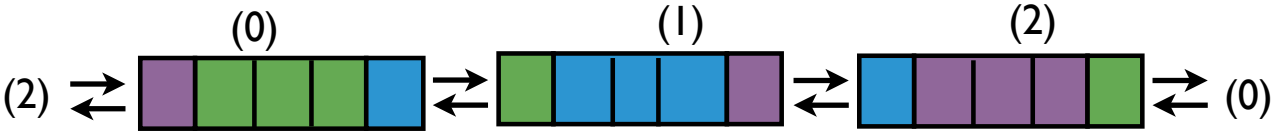
The Finite Differences (FD) example

Input data at rank 0

Scatter data for each participant

Each participant computes its finite differences

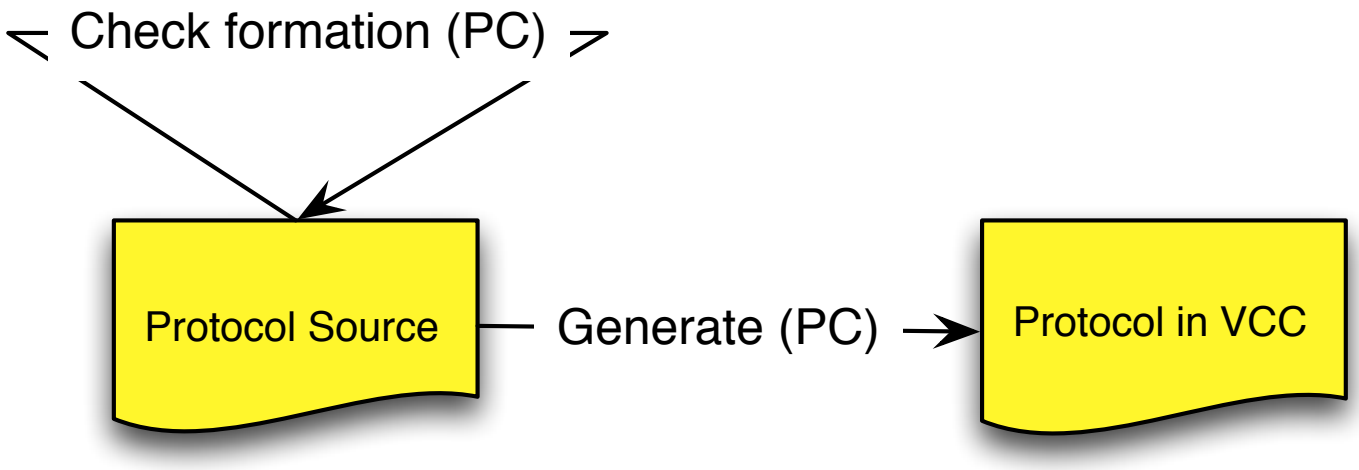
Send/recv boundary values to determine convergence



Perform global reduction (AllReduce) to compute the max error;
Loop if convergence criterion not met, up until to MAX_ITER

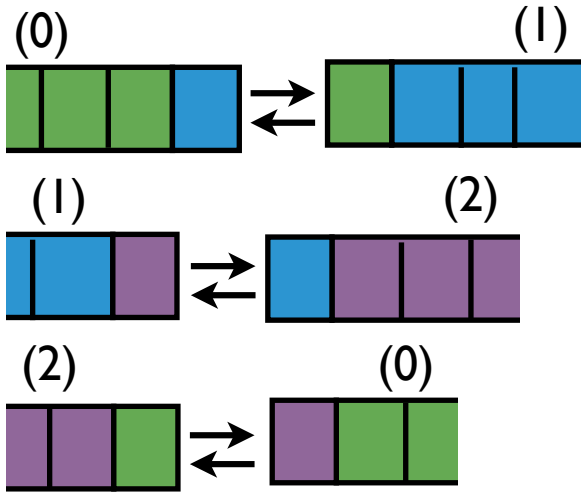
Gather the solution data at rank 0, if there was convergence.

First step: come up with a well-formed protocol



FD Protocol – Point-to-point comm.

message 0 1 **float**
message 1 0 **float**
message 1 2 **float**
message 2 1 **float**
message 2 0 **float**
message 0 2 **float**

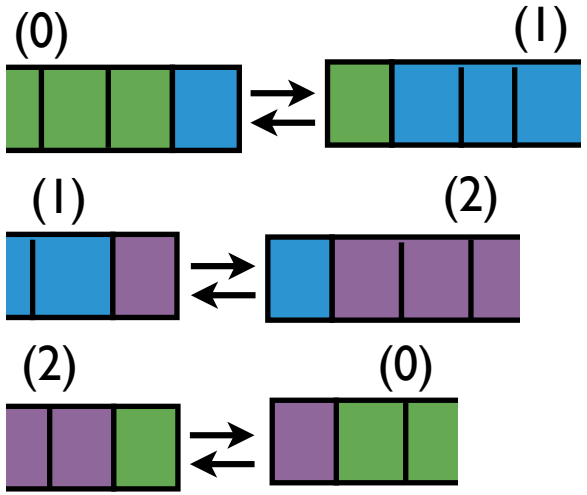


Source rank

Source rank

FD Protocol – Point-to-point comm.

message 0 1 **float**
message 1 0 **float**
message 1 2 **float**
message 2 1 **float**
message 2 0 **float**
message 0 2 **float**

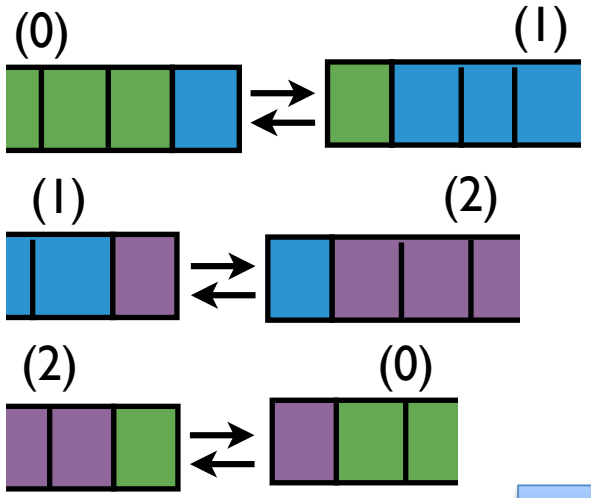



```
foreach i: 0 .. size-1 {
    // send right a floating point number
    message i (i=size-1 ? 0 : i+1) float
    // send left a floating point number
    message i (i=0 ? size-1 : i-1) float
}
```

Parameter of the protocol

FD Protocol – Point-to-point comm.

message 0 1 **float**
message 1 0 **float**
message 1 2 **float**
message 2 1 **float**
message 2 0 **float**
message 0 2 **float**



Parameter of the protocol

```
foreach iteration: 1 .. numIterations {
    foreach i: 0 .. size-1 {
        // send right a floating point number
        message i (i=size-1 ? 0 : i+1) float
        // send left a floating point number
        message i (i=0 ? size-1 : i-1) float
    }
}
```

FD Protocol – Collective comm.

Should be a multiple of the number of processes

// Process rank 0 divides the array among all processes
scatter 0 **float**[n]

… // point-to-point communication

// Each process proposes a floating point number;
// process rank 0 collects the max of these
reduce 0 **max float**
// Each process proposes its part of the solution,
// process rank 0 gathers these to form Xn, an array of length n
gather 0 **float**[n/**size**]

FD Protocol – Collective comm.

```
// The problem size must be a multiple of size, the number of processes
val numIterations: positive
// Process rank 0 broadcasts the maximum number of iterations
broadcast 0 n: {x: positive | x % size = 0}

// Process rank 0 divides the array among all processes
scatter 0 float[n]
...
ion

// Each process proposes a floating point number; process rank 0
// collects the max of these
reduce 0 max float
// Each process proposes its part of the solution,
// process rank 0 gathers these to form Xn, an array of length n
gather 0 float[n/size]
```

Introduces value **numIterations** in the protocol

Dependent Type specifying that **n** is a multiple of **size**

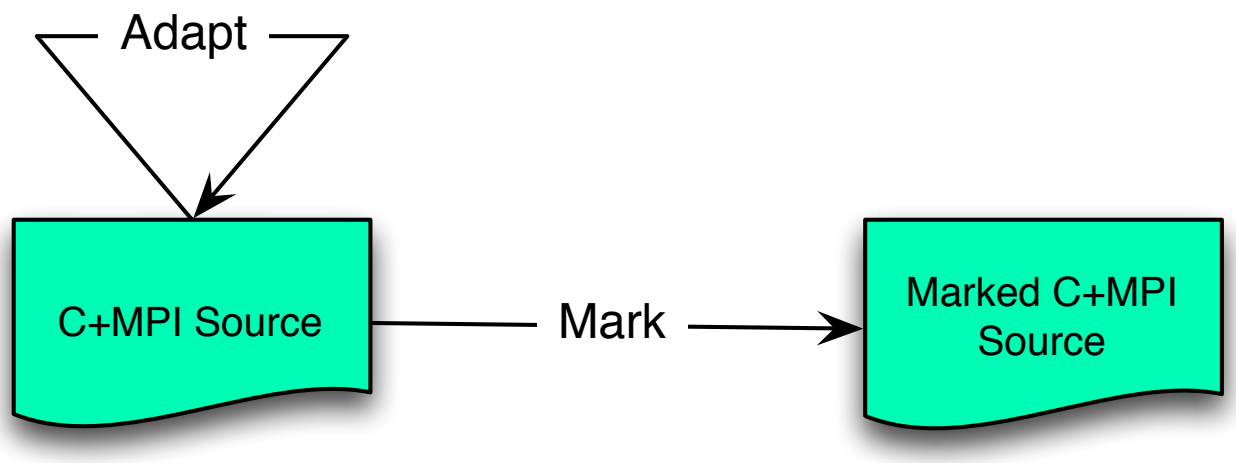
Demo time!

Let's play with an Eclipse plug-in that helps in writing well-formed protocol types

Also available online at:

http://gloss.di.fc.ul.pt/tryit/ParTypes

Second step:
Annotate the C+MPI code



Question:

- Does this C+MPI program follow the protocol?

Let's look at the code…

How do we annotate the code?!

1. We provide contracts for all MPI primitives
   - No need to annotate the primitives
2. We inject the Rank and the Size values
   - No need to explicit insert them in the protocol
3. Provide the actual value to insert for every val primitive
   - Annotate with _applyInt (or _applyIntArray…)

How do we annotate the code?!

4. Put a mark in the **for** loops that correspond to **foreach** in the protocol
   - The C language has only one for loop construct
   - The mark: _foreach
   - A tool expands it to VCC annotations
     - Clang LLVM plug-in not currently working ☹

Intuition about the for loop annotations

_foreach

```
for (iter=1; iter <= NUM_ITER; iter++) {
      // loop body
}
```

Expands to

```
_(ghost Cons _cfe0 = cons(_protocol, _rank))
_(ghost Protocol _fe0 = head(_cfe0))
_(assert foreachLower(_fe0) == 1)
_(assert foreachUpper(_fe0) == NUM_ITER)
_(ghost IntAbs _fe0Body = foreachBody(_fe0))
for (iter=1; iter <= NUM_ITER; iter++) {
    _(ghost _protocol = _fe0Body[iter])
    // loop body
    _(assert isSkip(_protocol,_rank))
 }
_(ghost _protocol = tail(_cfe0))
// Continuation of the program
```

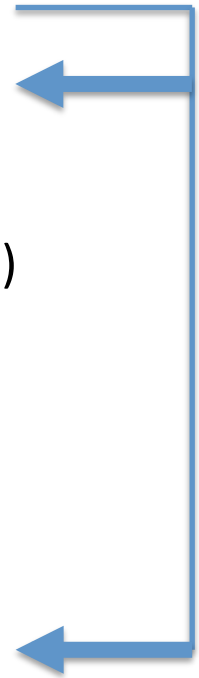
Intuition about the for loop annotations

_foreach

```
for (iter=1; iter <= NUM_ITER; iter++) {
      // loop body
}
```

Expands to

```
_(ghost Cons _cfe0 = cons(_protocol, _rank))
_(ghost Protocol _fe0 = head(_cfe0))
_(assert foreachLower(_fe0) == 1)
_(assert foreachUpper(_fe0) == NUM_ITER)
_(ghost IntAbs _fe0Body = foreachBody(_fe0))
for (iter=1; iter <= NUM_ITER; iter++) {
    _(ghost _protocol = _fe0Body[iter])
    // loop body
    _(assert isSkip(_protocol,_rank))
 }
_(ghost _protocol = tail(_cfe0))
// Continuation of the program
```



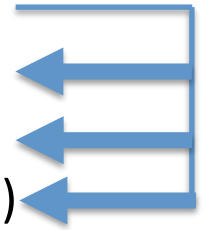
Intuition about the for loop annotations

_foreach

```
for (iter=1; iter <= NUM_ITER; iter++) {
      // loop body
}
```

Expands to

```
_(ghost Cons _cfe0 = cons(_protocol, _rank))
_(ghost Protocol _fe0 = head(_cfe0))
_(assert foreachLower(_fe0) == 1)
_(assert foreachUpper(_fe0) == NUM_ITER)
_(ghost IntAbs _fe0Body = foreachBody(_fe0))
for (iter=1; iter <= NUM_ITER; iter++) {
    _(ghost _protocol = _fe0Body[iter])
    // loop body
    _(assert isSkip(_protocol,_rank))
 }
_(ghost _protocol = tail(_cfe0))
// Continuation of the program
```



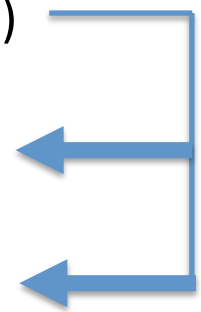
Intuition about the for loop annotations

_foreach

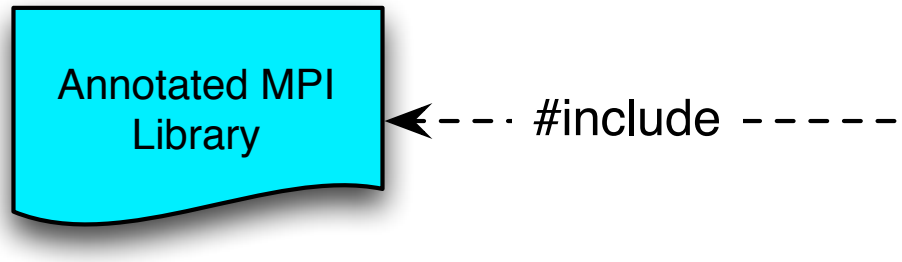
```
for (iter=1; iter <= NUM_ITER; iter++) {
      // loop body
}
```

Expands to

```
_(ghost Cons _cfe0 = cons(_protocol, _rank))
_(ghost Protocol _fe0 = head(_cfe0))
_(assert foreachLower(_fe0) == 1)
_(assert foreachUpper(_fe0) == NUM_ITER)
_(ghost IntAbs _fe0Body = foreachBody(_fe0))
for (iter=1; iter <= NUM_ITER; iter++) {
    _(ghost _protocol = _fe0Body[iter])
    // loop body
    _(assert isSkip(_protocol,_rank))
 }
_(ghost _protocol = tail(_cfe0))
// Continuation of the program
```



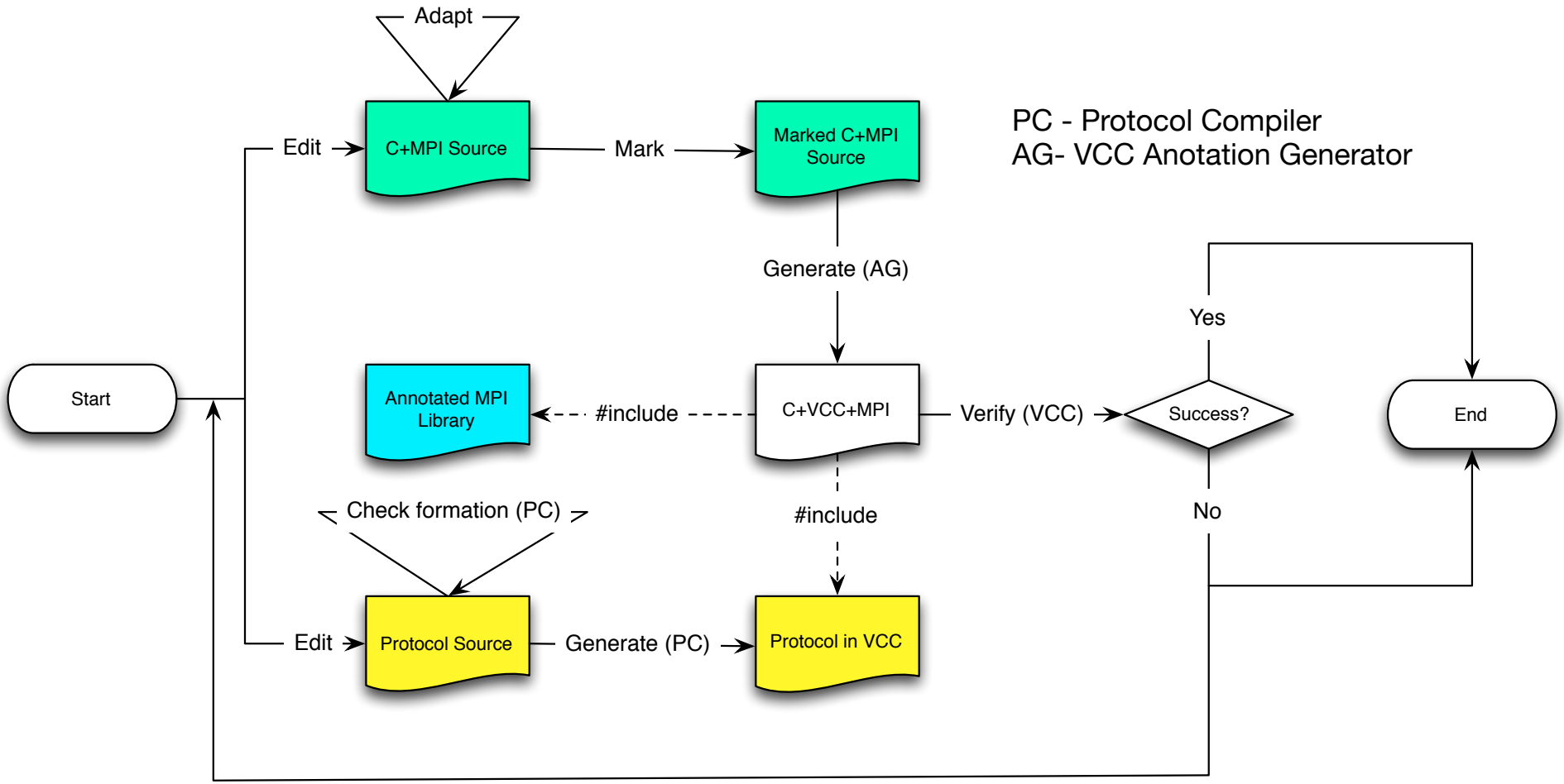
The ParTypes Theory



Eavesdrop VCC code

- MPI_InitAndFinalize.h
- MPI_CommRankAndSize.h
- MPI_SendAndRecv.h
- ParTypes.h

Our approach (Again)



Warning!

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth