Part1\_Compositions: General things about modules and composition

There are 3 examples containing modules composition. The first one is a temporal composition testing if a number is prime, the second one is a vertical composition for the same problem, while the third one is a diagonal composition for computing the minimum of 2 numbers.

You can open “agapia.txt” file from each example to see the code. You can also see that we used stdio library for reading/writing on console, which is being included by using file “Includes.h”.

There are two types of modules in AGAPIA:

* Atomic module - doesn't contain any AGAPIA composition or module invocation. This kind of module contains C/C++ code with some additions that we’ll see later.
* Coordination module – every module that is not atomic.

The entry point is the MAIN module which usually is a coordination module. The input for this module is specified in the “MainInput.txt” file (next chapter contains a real usage of this file).

I recommend open one of the examples before moving on.

Deployment: Right now, you can specify only if a module should execute on the master process or it is free to execute on whatever process the scheduler chooses. To hook a module execution to the master process use “@MASTER” after module declaration line. You’ll see later in examples this.

Rules for writing modules:

* Variables in module's interfaces must have only lowercase letters
* Variables must have different names in a module
* Allowed types for variables are int, float, char, buffer, string, bool and arrays of these.
* **Instead of using “%” operator to compute the module of two numbers, use “mod” (collides with spatial composition)**

About pointers (examples in next chapters)

You can use pointers in the program:

* In the iterative/shared memory model: send the address through interfaces, convert the address to an integer ( int \*p = something; out = (int) p; )
* In the distributed model: Use “buffer” type which provides automatic serialization when communicating between modules. It works for iterative/shared model too but it is not smart enough right now to avoid serialization in this case.

Part2: “FOR”, Array of processes, input for main module

Array of processes vs array of basic types

An array of integers is declared like arr : int[] while a vector of processes each one with a single integer, (nr : int ) numbers[].

* An array of basic data types can be sent as output/input to a single module while the reason to use vector of processes is to send/receive data to/from multiple modules like a scatter/gather.

Read/Write in an array of processes: if the declaration looks like ((a:int,b:int) array[]) then you can use it like this:

* WRITE: array@[index] = value; where “index” can be only identifier or integer number, and “value” can be one of the basic types or an identifier. It will not work in the current version to perform other operations (addition, multiply, increment, etc) on the same line, so you have to compute it before in another variable.
* You can take the values from a component like this:

SimpleProcessItem\* Name = array@[index]; OR VectorProcessItem\*Name= array@[index]; depending on what you have represented on the array.

To get a data item from a SimpleProcessItem you should call:

Name->GetItem(indexOfTheItem)->GetValue(); Example: if you gave a SimpleProcessItem like (a:int, b:float), then name->GetItem(1)->GetValue() will return the value of b.

There are some other complicated constructions for this like for buffers, but the examples in the next chapters will clarify this.

FOR statement

* Use FOREACH\_S/T/ST when you don't have dependencies between the body module for better performance
* In FOR\_S/T\_ST, the iteration variable can be only used now in high level AGAPIA statements, not inside modules - an example is having nested FORs, so you can use in the second FOR the iteration value from the first FOR.

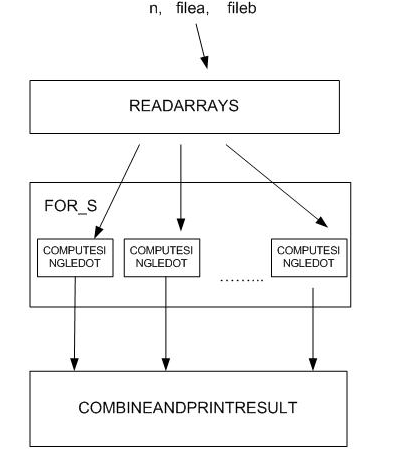
Example 0: SumOfNumbers

This example computes the sum of N numbers read from console. In READ module we fill a vector of processes, each process containing a single integer value.



1. Dot product of 2 vectors read from a file

The FOR\_EACH statement is used here.



This program computes each multiplication on a different process and joins them all in COMBINEANDPRINTRESULT. This module is an atomic one and it knows how many inputs it should wait because we are using a FOR module with a constant value. The JOIN operation is automatic in this case.

1. Normal FOR

This is just a simple program to demonstrate the functionality of FOR with a program inside which has dependencies between iterations.

Part3: Buffers

Part4: “IF” statement examples

The IF statement helps you coordinate input flows through different programs.

if (condition)

{

IF\_PROGRAM

}

else // optional !

{

ELSE\_PROGRAM

}

The IF\_PROGRAM and ELSE\_PROGRAM need to have exactly the same interfaces on inputs and outputs, and more than this, the same names for variables in the interfaces. If the else branch is missing, the IF\_PROGRAM should have the same west-east and north-south interfaces.

As you’ll see in the examples, you can use in the condition of IF the inputs for IF\_PROGRAM/ELSE\_PROGRAM and global variables available in the parent module.

There are four examples in this folder. The first three examples test if a number is odd or not. The last one is combining “for\_each” and “if” statements to convert all numbers in an array from even to odd.

Part5: “WHILE” statement examples

There are 3 simple examples here which all output the divisors of 3 smaller than an input number. The same rules as for the “IF” statement applies here too, excepting that if we a WHILE like:

while\_s/t/st (condition)

{

PROGRAM

}

, then PROGRAM must have the same north-south interface for while\_t, same east-west interface for while\_s and same north-south, east-west interfaces for while\_st.

Part6: Hard problems

This section contains now just 2 examples solving the edge detection problem by using Sobel filter. There is an iterative implementation which uses pointers, and another version which works for both distributed and shared memory and uses buffers. The differences between them are minimal.