1. Bindings

In addition to Java, COMPSs supports the execution of applications written in other languages by means of bindings. A binding manages the interaction of the application with the COMPSs Java runtime, providing the necessary language translation.

The next subsections describe the language bindings provided by COMPSs.

1.1. Python

COMPSs features a binding for Python 2.x applications. The next subsections explain how to program a Python application for COMPSs and how to configure the binding library.

1.1.1. Programming Model

1.1.1.1. Task Selection

Like in the case of the Java language, a COMPSs Python application is a sequential program that contains calls to tasks. In particular, the user can select as a task:

- Functions
- Instance methods: methods invoked on objects.
- Class methods: static methods belonging to a class.

Regarding task selection, in Python it is not done by means of an annotated interface but with the use of Python decorators. In particular, the user needs to add, before the definition of the function/method, a **@task** decorator that describes the task.

As an example, let us assume that the application calls a function *func*, which receives a string parameter containing a file name and an integer parameter. The code of *func* updates the file.

```
my_file = 'sample_file.txt'
func(my_file, 1)
```

In order to select *func* as a task, the corresponding *@task* decorator needs to be placed right before the definition of the function, providing some metadata about the parameters of that function. The metadata corresponding to a parameter is specified as an argument of the decorator, whose name is the formal parameter's name and whose value defines the type and direction of the parameter. The parameter types and directions can be:

- Types*: primitive types* (integer, long, float, boolean), *strings*, *objects* (instances of user-defined classes, dictionaries, lists, tuples, complex numbers) and *files* are supported.
- Direction: it can be read-only (*IN* - default), read-write (*INOUT*) or write-only (*OUT*).

COMPSs is able to automatically infer the parameter type for primitive types, strings and objects, while the user needs to specify it for files. On the other hand, the direction is only mandatory for *INOUT* and *OUT* parameters. Thus, when defining the parameter metadata in the *@task* decorator, the user has the following options:

- *INOUT*: the parameter is read-write. The type will be inferred.
- *OUT*: the parameter is write-only. The type will be inferred.
- *FILE*: the parameter is a file. The direction is assumed to be *IN*.
- *FILE_INOUT:* the parameter is a read-write file.
- *FILE_OUT:* the parameter is a write-only file.

Consequently, please note that in the following cases there is no need to include an argument in the *@task* decorator for a given task parameter:

- Parameters of primitive types (integer, long, float, boolean) and strings: the type of these parameters can be automatically inferred by COMPSs, and their direction is always *IN*.
- Read-only object parameters: the type of the parameter is automatically inferred, and the direction defaults to *IN*.

Continuing with the example, in the following code snippet the decorator specifies that *func* has a parameter called *fi*, of type *FILE* and *INOUT* direction. Note how the second parameter, *i*, does not need to be specified, since its type (integer) and direction (*IN*) are automatically inferred by COMPSs.

```
from pycompss.api.task import task
from pycompss.api.parameter import *

@task(f = FILE_INOUT)
def func(f, i):
    fd = open(f, 'r+')
    …
```

If the function or method returns a value, the programmer must specify the type of that value using the *returns* argument of the *@task* decorator:

```
@task(returns = int)
def ret_func():
    return 1
```

For tasks corresponding to instance methods, by default the task is assumed to modify the callee object (the object on which the method is invoked). The programmer can tell otherwise by setting the *isModifier* argument of the *@task* decorator to *False*.

```
class MyClass(object):
    …

    @task(isModifier = False)
    def instance_method(self):
        …  # self is NOT modified here
```

The programmer can also mark a task as a high-priority task with the *priority* argument of the *@task* decorator. This way, when the task is free of dependencies, it will be scheduled before any of the available low-priority (regular) tasks. This functionality is useful for tasks that are in the critical path of the application's task dependency graph.

```
@task(priority = True)
def func():
    …
```

Table 1 summarizes the arguments that can be found in the *@task* decorator.

Table 1 – Arguments of the @task decorator

Argument	Value
Formal parameter name	– `INOUT`: read-write parameter, all types except file (primitives, strings, objects). – `OUT`: read-write parameter, all types except file (primitives, strings, objects). – `FILE`: read-only file parameter. – `FILE_INOUT`: read-write file parameter. – `FILE_OUT`: write-only file parameter.
`returns`	`int` (for integer and boolean), `long`, `float`, `str`, `dict`, `list`, `tuple`, user-defined classes
`isModifier`	`True` (default) or `False`
`priority`	`True` or `False` (default)

1.1.1.2. Main Program

The main program of the application is a sequential code that contains calls to the selected tasks. In addition, when synchronizing for task data from the main program, there exist two API functions that need to be invoked:

- *compss_open(file_name, mode = 'r')*: similar to the Python *open()* call. It synchronizes for the last version of file *file_name* and returns the file descriptor for that synchronized file. It can receive an optional parameter *mode*, which defaults to *'r'*, containing the mode in which the file will be opened (the open modes are analogous to those of Python *open()*).
- *compss_wait_on(obj, to_write = True)*: synchronizes for the last version of object *obj* and returns the synchronized object. It can receive an optional boolean parameter *to_write*, which defaults to *True*, that indicates whether the main program will modify the returned object.

To illustrate the use of the aforementioned API functions, the following example first invokes a task *func* that writes a file, which is later synchronized by calling *compss_open()*. Later in the program, an object of class *MyClass* is created and a task method *method* that modifies the object is invoked on it; the object is then synchronized with *compss_wait_on()*, so that it can be used in the main program from that point on.

```
from pycompss.api.api import compss_open, compss_wait_on

my_file = 'file.txt'
func(my_file)
fd = compss_open(my_file)
…

my_obj = MyClass()
my_obj.method()
my_obj = compss_wait_on(my_obj)
…
```

The corresponding task selection for the example above would be:

```
@task(f = FILE_OUT)
def func(f):
    …


class MyClass(object):
    …

    @task()
    def method(self):
        … # self is modified here
```

Table 2 summarizes the API functions to be used in the main program of a COMPSs Python application.

Table 2 – COMPSs Python API functions

Function	Use
`compss_open(file_name, mode = 'r')`	Synchronizes for the last version of a file and returns its file descriptor.
`compss_wait_on(obj, to_write = True)`	Synchronizes for the last version of an object and returns it.

1.1.1.2.1. Future Objects

If the programmer selects as a task a function or method that returns a value, that value is not generated until the task executes. However, in order to keep the asynchrony of the task invocation, COMPSs manages *future objects*: a representant object is immediately returned to the main program when a task is invoked.

```
@task(returns = MyClass)
def ret_func():
    return MyClass(…)

…

# o is a future object
o = ret_func()
```

The future object returned can be involved in a subsequent task call, and the COMPSs runtime will automatically find the corresponding data dependency. In the following example, the future object *o* is passed as a parameter and callee of two subsequent (asynchronous) tasks, respectively:

```
# o is a future object
o = ret_func()

…

another_task(o)

…

o.yet_another_task()
```

In order to synchronize the future object from the main program, the programmer proceeds in the same way as with any object updated by a task:

```
# o is a future object
o = ret_func()

…

o = compss_wait_on(o)
```

The future object mechanism is applied to primitive types, strings and objects (including the Python built-in types list, dictionary, tuple and complex).

It is important to note that, for instances of user-defined classes, the classes of these objects should have an empty constructor, otherwise the programmer will not be able to invoke task instance methods on those objects:

```
class MyClass(object):
    def __init__(self): # empty constructor
       …

    …

o = ret_func()

# invoking a task instance method on a future object can only be done
# when an empty constructor is defined in the object's class
o.yet_another_task()
```

1.1.1.3. Important Notes

For the COMPSs Python binding to function correctly, the programmer should not use relative imports in her code. Relative imports can lead to ambiguous code and they are discouraged in Python, as explained in:

```
http://docs.python.org/2/faq/programming.html#what-are-the-best-practices-
for-using-import-in-a-module
```

1.1.2. Execution

1.1.2.1. Environment

The following environment variables must be defined before executing a COMPSs Python application:

- *JAVA_HOME*: Java JDK installation directory (e.g. /usr/lib/jvm/java-6-openjdk/)

1.1.2.2. Command

In order to run a Python application with COMPSs, the script *runcompssext* can be used. An example of an invocation of the script is:

```
> runcompssext
--lang=python
--app=$TEST_DIR/test.py
--classpath=$TEST_DIR
--library_path=/home/user/libdir
--cline_args="arg1 arg2"
--project=$TEST_DIR/project.xml
--resources=$TEST_DIR/resources.xml
--tracing=true
```

The options of the script are:

- --lang=python
- --app=<path>: path to the .py file containing the main program.
- --classpath=<path>: path/s where to search for the application's Python modules. The default value is the current directory.
- --library_path=<path>: path/s where to search for libraries that are not in a standard path. The default value is the variable $LD_LIBRARY_PATH.
- --cline_args=<args>: arguments to pass to the application.
- --project=<proj_file>: path of the project XML file.
- --resources=<res_file>: path of the resources XML file.
- --tracing=<true | false>: generate execution traces. Default is false.