RAVEN interaction with External Applications

RAVEN Workshop

PSA 2015 - April 26th 2015, Sun Valley (ID)

www.inl.gov

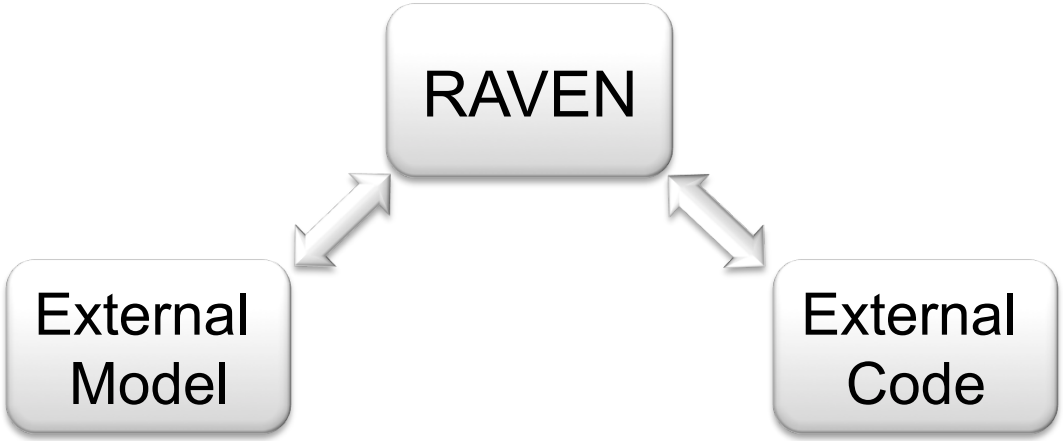
Idaho National Laboratory

Outline

- Overview of RAVEN interaction with external Applications
  - Available APIs: External Model and Code APIs
- Using the External Model Entity
  - Introduction
  - Implementing the `Python` module
  - Available methods
  - Interaction with RAVEN
- Coupling a new Application through a Code Interface
  - Introduction
  - Code requirements
  - Interfaces that need to be implemented
  - Interaction with RAVEN

RAVEN Interaction with External Applications

- RAVEN has two preferential APIs to interact with external Applications
  - External Model: An external Python “entity” that can act as, for example, a system model
  - External Code: API to drive external system codes
- Both APIs are written in `PYTHON`

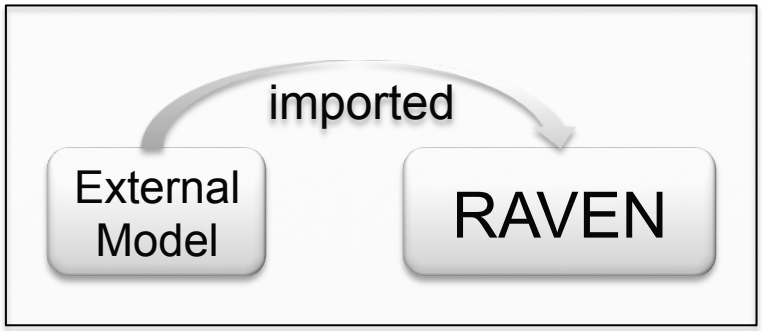


External Model

External Model Entity: Introduction

- The *ExternalModel* object represents an entity that is embedded in RAVEN at runtime
- This object allows the user to create a `PYTHON` module that is going to be treated as a pre-defined internal Model

It becomes part of the RAVEN framework



External Model Entity: RAVEN Input

```
<Models>
  <ExternalModel name='PythonModule' subType='' ModuleToLoad='workshop_model'>
       <variable>x1</variable>
       <variable>y1</variable>
       <newCustomXmlNode>
         <printString>
          We are not here
          to brush dolls
         </printString>
       </newCustomXmlNode>
  </ExternalModel>
</Models>
```

Model variables tracked by RAVEN

File name (with or without path): workshop_model.py

Variables that RAVEN is able to “see” and “check”

Custom piece of XML Input

External Model Entity: `Python` Module

- In the External `Python` module, the user can implement all the methods that are needed for the functionality of the model
- Only these methods are called by the framework:

```
def _readMoreXML(self,xmlNode)
```
Optional

```
def initialize(self,runInfo,inputs)
```
Optional

```
def createNewInput(self,inputs,samplerType,**Kwargs)
```
Optional

```
def run(self,inputs)
```
Required

Each variable defined in the XML input is available in "self"

```
y1 = self.x1
```

External Model Entity: _readMoreXML

- Needed only if the XML input that belongs to the External Model needs to be extended to contain other information
- Input data needs to be stored in “self” in order to be available to all the other methods

```
def _readMoreXML(self,xmlNode):
  # get the node
  ourNode = xmlNode.find('newCustomXmlNode')
  # get the information in the node
  self.ourNewVariable = ourNode.text
```

Optional

<newCustomXmlNode> is unknown (in RAVEN)

External Model Entity: initialize

- This method initializes the model
- For example, it can be used to compute a quantity needed by the “run” method:

```
def initialize(self,runInfo,inputs):
  # initialize some quantities
  self.sigma = 10.0
  self.rho   = 5.0
```

List of Inputs defined in the Step entity

Optional

Info read in <RunInfo> XML input

Variables not listed in the XML input files

External Model Entity: createNewInput

- Can be used to create a custom Input with the information coming from RAVEN
- The generated input is transferred to the “run” method

Type of Sampler

Container of Info coming from RAVEN

```
def createNewInput(self,inputs,samplerType,**Kwargs):
  if samplerType == 'MonteCarlo':
    aVariable = inputs['anInput']*self.sigma
  else:
    aVariable = inputs['anInput']*self.rho
  aLocalVar = Kwargs['SampledVars']['x1']
  returnVar = aVariable*aLocalVar
  return returnVar
```

Optional

From initialize method

External Model Entity: run

- In this function, the user needs to implement the algorithm that RAVEN will execute
- The *run* method is generally called after having inquired the *createNewInput* method (internal or the user-implemented)

```
def run(self,inputs):
  input = inputs[0]
  self.y1 = returnVar**2
  return
```

List of Inputs generated in creatNewInput method

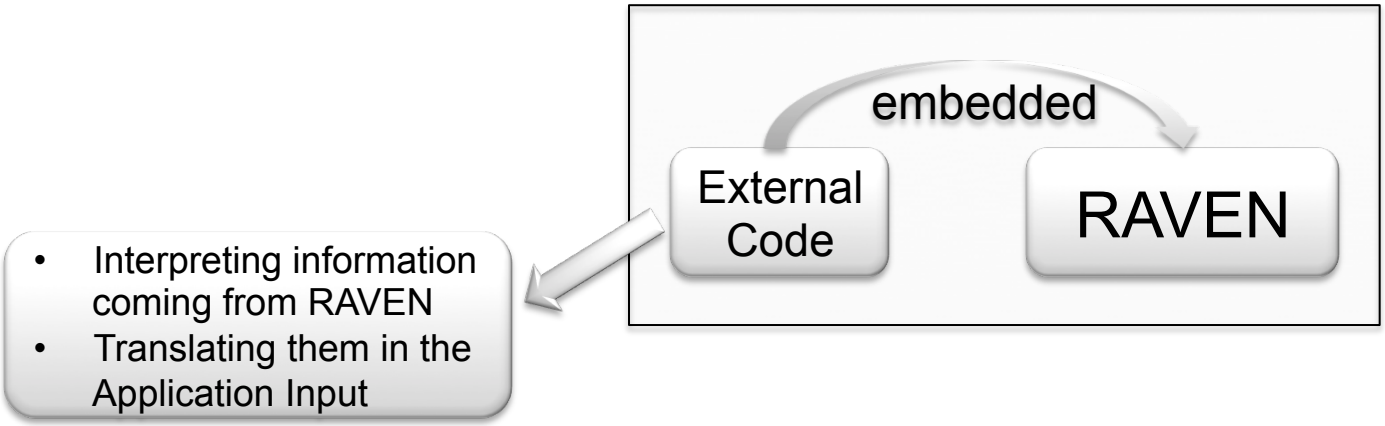
Required

Outcome stored in "self" => RAVEN can collect it

Code Interface

Coupling an Application with RAVEN: Introduction

- The procedure of coupling a new Application with RAVEN is a straightforward process
- The coupling is performed through a `Python` Interface
- The Interface has two functions:
  1. Interpret the information coming from RAVEN
  2. Translate such information in the input of the system code
- The coupling procedure does not require any modification of RAVEN



Coupling an Application with RAVEN: Requirements

- Input requirements:
  - Python-compatible parser for Application input
  - Decide the syntax your Code Interface will be able to interpret

Input Parser Example

```
class simpleInputParser():
  def __init__(self,filen):
    self.keyDict = {}
    lines=open(filen).readlines()
    for line in lines:
      key=line.split("=”)[0]
      value=line.split("=”)[1]
      self.keyDict[key]=value
  def modifyInternalDict(self,inDictionary):
    for key,newvalue in inDictionary.items():
      self.keyDict[key]=newvalue
  def writeNewInput(self,filen):
    fileobject = open(filen)
    for key,value in self.keyDict.items():
      fileobject.write(key+‘‘=’’+str(value)+‘‘\n’’)
```

Input Text

```
Key1 = aValue1
Key2 = aValue2
Key3 = aValue3
```

Coupling an Application with RAVEN: Requirements

- Output requirements:
  - RAVEN handles Comma Separated Values (CSV) files
  - If your code output is not in CSV format, your interface needs to convert it into CSV format

Input Parser Example

```
def convertOutputFileToCSV(outfile):
  keywordDict = {}
  fileobject = open(outputfile)
  outputCSVfile = open (outputfile + '.csv')
  lines = fileobject.readlines()
  for line in lines:
    listSplitted = line.split("=")
    keyword = listSplitted[0]
    value = listSplitted[1]
    keyDict[keyword] = value
  outputCSVfile.write(','.join(keyDict.keys()))
  outputCSVfile.write(','.join(keyDict.values()))
```

Output Text

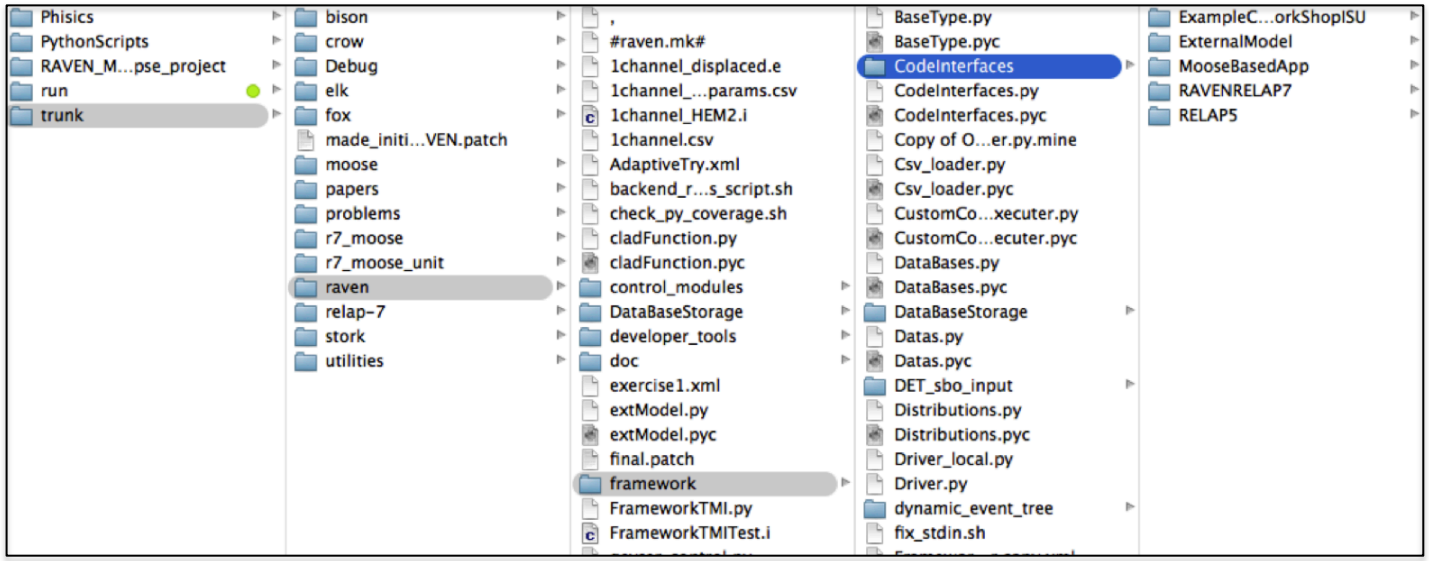
```
result1= aValue1
result2= aValue2
```

Output CSV

```
result1,result2
aValue1,aValue2
```

Coupling an Application with RAVEN: Interfaces

- RAVEN becomes aware of the codes it can use as Models only at run-time
  - RAVEN looks for code interfaces and loads them automatically
- The code interface needs to be placed in a new folder under the directory "*./raven/framework/CodeInterfaces*"



Coupling an Application with RAVEN: Methods

- RAVEN imports all the "Code Interfaces" at run-time, without actually knowing the syntax of the driven codes
- In order to make RAVEN able to drive a new Application, a `Python` module containing few methods (strict syntax) needs to be implemented:

```
class newApplication(CodeInterfaceBase):
```

`def generateCommand(self,input,exe,clargs,fargs)`	Required
`def createNewInput(self,inputs,samplerType,**Kwargs)`	Required
`def finalizeCodeOutput(self,command,output,workDir)`	Optional
`def checkForOutputFailure(self,output,workDir)`	Optional
`def getInputExtension(self)`	Optional

Coupling an Application with RAVEN: generateCommand

- Used to retrieve the command needed to launch the driven App and the root of the output file
- The return data type must be a TUPLE

List of input files

Optional Arguments

```
def generateCommand(self,input,exe,clargs,fargs):
  inp = input[0]
  outfile='out~'+os.path.split(inp)[1].split('.')[0]
  executeCommand = (exe+ ' -i ' +os.path.split(inp)[1])
  return (executeCommand,outfile)
```

Required

Executable

Coupling an Application with RAVEN: createNewInput

- Used to generate an input based on the information that RAVEN passes
- This method needs to return a list containing the path and filenames of the modified input files

List of inputs

Type of Sampler

Dictionary of info from RAVEN

```
def createNewInput(self,inputs,samplerType,**Kwargs):
  parser = simpleInputParser(currentInputFiles[0])
  sampledVars = Kwargs['SampledVars']
  parser.modifyInternalDictionary(sampledVars)
  temp = str(inputs[index][:])
  newInputFiles = copy.copy(inputs)
  newInputFiles[0] = os.path.join(os.path.split(temp)
    [0], Kwargs['prefix']+"~"+os.path.split(temp)[1])
  parser.writeNewInput(newInputFiles[0])
  return newInputFiles
```

Required

Coupling an Application with RAVEN: finalizeCodeOutput

- Used to convert the whatever App output format into a CSV
- RAVEN checks if a string is returned
  - RAVEN interprets that string as the new output file name (CSV)

Command created by ***generateCommand***

```
def finalizeCodeOutput(self,command,output,workDir):
  outputfile = os.path.join(workingDir,output+'.o')
  convertOutputFileToCSV(outfile)
```

Optional

Output root name

Working directory

Thank you

Questions?

Appendix: `Python`

Brief introduction to `Python`: why `Python`?

- Natural Language Tool-Kit
- Ease of use => interpreter
- AI Processing: Symbolic
  - `Python`'s built-in datatypes for strings, lists, and more
  - `Java` or `C++` require the use of special classes for this
- AI Processing: Statistical
  - `Python` has strong numeric processing capabilities: matrix operations, etc.
  - Suitable for probability and machine learning code

Brief introduction to `Python`: Example

```
x = 34 - 23                # A comment.
y = “Hello”                # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World” # String concat.
print x
print y
```

Brief introduction to `Python`: Basic knowledge

- Assignment uses = and comparison uses ==
- For numbers +-*/% are as expected
  - Special use of + for string concatenation
  - Special use of % for string formatting
- Logical operators are words (**`and, or, not`**)
  not symbols (&&, ||, !)
- The basic printing command is “print”
- First assignment to a variable will create it
  - Variable types don’t need to be declared
  - `Python` figures out the variable types on its own

Brief introduction to `Python`: datatypes

- Integers (default for numbers)
  - z = 5 / 2 # Answer is 2, integer division
- Floats
  - x = 3.456
- Strings
  - Can use “” or ‘’ to specify. “abc” ‘abc’ (Same thing)
  - Unmatched ones can occur within the string. “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them: “““a‘b“c”””
- Lists
  - Lists are unsorted array of objects (floats, integers, derived types, etc.)
  - a = [“Hello”,5.0,1]
- Dictionaries
  - Dictionaries are objects that employ a mapping between string keys and objects
  - a = {‘a’:1.0,’b’:’Hello’,’c’:1}

Brief introduction to `Python`: whitespace

- Whitespace is meaningful in `Python`: especially indentation and placement of newlines
  - Use a newline to end a line of code
    (Not a semicolon like in `C++` or `Java`)
    (Use \ when must go to next line prematurely)
  - No braces { } to mark blocks of code in `Python`…
    Use consistent indentation instead. The first line with a new indentation is considered outside of the block
  - Often a colon appears at the start of a new block. (We'll see this later for function and class definitions.)

Brief introduction to `Python`: Comments

- Start comments with # – the rest of line is ignored.
- Can include a “documentation string” as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it’s good style to include one.

```
def my_function(x, y):
  “““This is the docstring. This
  function does blah blah blah.”””
  # The code would go here...
```

Brief introduction to `Python`: example

```
x = 34 - 23                # A comment.
y = “Hello”                # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”       # String concat.
print x
print y
```

Brief introduction to `Python`: `Python` and Types

`Python` determines the data types in a program automatically — "Dynamic Typing"

But `Python`'s not casual about types, it enforces them after it figures them out — "Strong Typing"

So, for example, you can't just append an integer to a string. You must first convert the integer to a string itself

```
x = "the answer is " # Decides x is string.
y = 23               # Decides y is integer.
print x + y   # Python will complain about this.
```

Brief introduction to `Python`: Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

  `bob  Bob  _bob  _2_bob_  bob_2  BoB`

- There are some reserved words:

  `and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while`