Wrapping an SST simulation in OCCAM

Be on the lookout for this fellow: The call-outs are ACTIONs for you to do!

When you see the check mark, compare your work to the marked element
Objectives

In this exercise you will

• Learn how to use OCCAM
  ◦ Taking a role of a software developer

• Learn how to encapsulate an SST simulation
  ◦ You’re provided with a ready to run simulation
  ◦ You’ll learn how to run it
  ◦ You’ll encapsulate it in an OCCAM object
Outline

① The need for OCCAM
  ◦ Why encapsulate simulations

② Overview of the simulation
  ◦ Learn how it is built and run
  ◦ Learn its inputs and configurations

③ Encapsulate the simulation in OCCAM
  ◦ Configuration options
  ◦ Inputs and outputs
  ◦ Build and run scripts
The need for encapsulation
The need for encapsulation

The problem

- Simulators are heterogeneous
  - Different ways of building, running

- Heterogeneity makes interoperability difficult
  - How are outputs passed as input to another program

- Requires users extra knowledge
  - How to build, run, and configure
① The need for encapsulation

Our solution

- Encapsulating software abstracts heterogeneity
  - Use standard interfaces
- Provide the means to describe software
  - Build/run scripts
  - Specifying inputs/outputs
  - Specifying configurations

How does that help?
The need for encapsulation

Our solution

- OCCAM objects are composable
  - Are agnostic to contents
  - Are easily connected
  - Types and schemas allow well-formed composition
Simulating a simple SST-based CPU
In this exercise you’ll take a role of a developer

- You developed a simulator (XSim)
  - Simplified MIPS-like architecture
- Your simulator leverages the SST framework
  - Uses SST framework for simulation
  - Uses SST’s memory simulator
Simulating an SST-based CPU

Introduction

Once you complete this section you will

1. Be familiar with the location of software
   ◦ Code, configuration files, inputs

2. Be able to run a simulation using Xsim
   ◦ How to pass configurations and inputs

But… Why do I need to know this?
Simulating an SST-based CPU

Introduction

When encapsulating a simulator you need to know how to use the simulator.
Simulating an SST-based CPU

The simulation files

In your machine you have

1. SST framework pre-installed
2. The source code for XSim
3. The SST simulation description
4. A benchmark program (input to Xsim)
5. The simulation parameters (configuration)
Simulating an SST-based CPU

The simulation files

(a) Follow the instructions in the handout on terminal access

(b) navigate to exercise 2 folder
Simulating an SST-based CPU

The simulation files

(a) Run ‘ls’ to view the contents

```
user@tutorial ~/exercises/ex2 $ ls
auxiliary_materials XSim
user@tutorial ~/exercises/ex2 $ ls auxiliary_materials
default_program.m joinoutputs.py runner.py simulation.py
default_simulation.json occam.py sim_parser.py
user@tutorial ~/exercises/ex2 $
```

(b) Run ‘ls auxiliary_materials’ to view the contents

We’ll look into the contents of the auxiliary materials folder in a moment
Simulating an SST-based CPU

Building XSim

First we need to build XSim

- XSim is implemented in C++
- Is built using CMake
  - Generates build scripts for different platforms
Simulating an SST-based CPU

Building XSim

First create and enter a build directory:

- This keeps the code folder clean

(a) Create a build directory

(b) Enter the directory

```
user@tutorial ~/exercises/ex2 $ mkdir build
user@tutorial ~/exercises/ex2 $ cd build
user@tutorial ~/exercises/ex2/build $ 
```
Exercise materials

Building XSim

Executing CMake will generate a Makefile

(a) Run cmake command
② Simulating an SST-based CPU

Building XSim

Running make will compile the code

(a) Run the make command

```
user@tutorial ~/exercises/ex2/build $ make
Scanning dependencies of target XSim_instruction_set
[ 2%] Building CXX object CMakeFiles/XSim_instruction_set.dir/implementation/src/mips_core_simulator/instruction_set/instruction_types/raw_type_t.cpp.o
[ 4%] Building CXX object CMakeFiles/XSim_instruction_set.dir/implementation/src/mips_core_simulator/instruction_set/instruction_types/r_type_t.cpp.o
[ 6%] Building CXX object CMakeFiles/XSim_instruction_set.dir/implementation/src/mips_core_simulator/instruction_set/instruction_types/i_type_t.cpp.o
[ 9%] Building CXX object CMakeFiles/XSim_instruction_set.dir/implementation/src/mips_core_simulator/instruction_set/instruction_types/ix_type_t.cpp.o
```
Simulating an SST-based CPU

Building XSim

The compiled simulator is in the build directory

```
user@tutorial ~/exercises/ex2/build $ ls
ARGS    cmake_install.cmake
CMakeCache.txt  libXSim_CORE.so
CMakeFiles  libXSim_instruction_set.so
user@tutorial ~/exercises/ex2/build $ ls
libXSim_memory.so  Makefile
libXSim.so
libXSim_state.so
```

Check your output
Simulating an SST-based CPU

The simulation description

Next let's see the SST description

(a) Find and open the file simulation.py in the auxiliary materials folder

```
user@tutorial ~/exercises/ex2/auxiliary_materials $ ls
default_program.m join_outputs.py runner.py
default_simulation.json occam.py sim_parser.py
user@tutorial ~/exercises/ex2/auxiliary_materials $
```
Simulating an SST-based CPU

The simulation description

The simulation setup using the SST API

```python
import sst
import sim_parser

arguments = sim_parser.parse()

cpu = sst.Component("XSim", "XSim.core")
cpu.addParams(arguments.get_cpu_params())

memory = sst.Component("memory", "memHierarchy.MemController")
memory.addParams(arguments.get_memory_params())

cpu_memory_link = sst.Link("cpu_memory_link")
cpu_memory_link.connect(
    ( cpu, "data_memory_link", arguments.get_link_latency() ),
    ( memory, "direct_link", arguments.get_link_latency() )
)
```

Parse inputs

Create and configure CPU

Create and configure memory

Connect cpu and memory
Simulating an SST-based CPU

The simulation input

XSim takes a benchmark program as input

- SST simulation description
- Program
- Simulation parameters

(a) Find and open the file default_program.m in the auxiliary materials folder
Simulating an SST-based CPU

The simulation input

A document with one 16-bit instruction per line

Instructions

0x81BB  #0  liz $r1, 187  |  int stop(r1)=187;
0x8201  #1  liz $r2, 1   |  int step(r2)=1;
0x8000  #2  liz $r0, 0   |  for(int i(r0)=0; (r1-r0)==0; r0=r0+r2)
0x0B20  #3  sub $r3, $r1, $r0  |
0xBB08  #4  bz $r3, 8     |
0x4800  #5  sw $r0, $r0   |
0x0008  #6  add $r0, $r0, $r2 |
0xC003  #7  j 3          |

# starts a comment
Simulating an SST-based CPU

The simulation is configured using a JSON file.

(a) Find and open the file default_simulation.m in the auxiliary materials folder.
Simulating an SST-based CPU

The simulation parameters

```json
{
  "cpu": {
    "clock_frequency": "2MHz"
  },
  "memory": {
    "backend": "memHierarchy.simpleMem",
    "backend.mem_size": "2MiB",
    "clock": "200Hz",
    "simpleMem": {
      "access_time": "100ns"
    }
  }
}
```
Simulating an SST-based CPU

Running the simulation

The simulation is executed by invoking:

```
python runner.py
--simulation <simulation file>
--build-folder <path to build folder>
--input-program <file>
--simulation-config <file>
--output-file <file>
```

(a) Run the simulation
② Simulating an SST-based CPU

The output

The output is now in the current folder

(a) Find and open the file statistics.json in the auxiliary materials folder

```bash
user@tutorial ~/exercises/ex2/auxiliary_materials $ ls
default_program.m join_outputs.py runner.py simulation.py
default_simulation.json occam.py sim_parser.py statistics.json
user@tutorial ~/exercises/ex2/auxiliary_materials $
Simulating an SST-based CPU

The output

Output contains info. about the CPU execution

```json
{"experiments": [{
  "registers": [{
    "r0": 187, "r1": 187, "r2": 1, "r3": 0, "r4": 0, "r5": 0, "r6": 0, "r7": 17391}],
  "stats": [{
    "add": 561, "bz": 376, "cycles": 3740012, "halt": 1, "instructions": 2070, "j": 374, "liz": 7, "lw": 187, "put": 1, "sub": 376, "sw": 187}],
  "clock_frequency": "200Hz",
  "dataset": "Simulation Name"
}...]
```
Wrapping the simulation in OCCAM
Wrapping the simulation

**Interface with OCCAM**

- Wrapping the simulator in OCCAM
  - Building/running becomes transparent for the user
  - Makes it interoperable with other OCCAM objs.

- Wrapping requires implementing interfaces
  - Schemas and scripts
  - Simple to implement
  - But may require some effort from the developer

- Wrap once, use freely
  - Wrapping is a one time effort
  - Users use the wrapped simulator
You’ll perform 4 steps to create the interfaces:

1. Create a new OCCAM object with XSim
2. Implement build and run scripts
3. Implement input & output interfaces
4. Implement configuration schema

Simulation

SST Framework

Memory Simulator

XSim CPU Simulator

SST simulation description

Program

Simulation parameters

Input

Output

Output Schema

Build/Run Scripts

XSim

Configuration

Configuration Schema
Create a new OCCAM object

- Syntax: `occam new <type> <name>
- Creates a skeleton of an object
  - In a directory named `<type>-<name>`

(a) Go to the ex2 directory

(b) Create a new OCCAM simulator

(c) Enter the created directory
In the new directory you’ll find

- A git repository (.git directory)
- The object specification (object.json)

Why do I need these?
Wrapping the simulation

Step 1 – Create the object

The git repository

- Tracks the contents of the object
  - Manages which files belong to the object
  - Tracks changes in each file

- Allows to revert to a previous version
  - Old versions can be used and inspected
  - Even if the object is changed
  - Allows tracking provenance of generated objects
Wrapping the simulation

Step 1 – Create the object

The object specification

- Contains object metadata
  - Metadata for OCCAM
  - Other user provided metadata
- We will build-up this file as we progress
  - Input/output/configurations declared here

object.json

```json
{
  "name": <name>,
  "id": <generated UUID>,
  "type": "simulator"
}
```

Check your object.json
Wrapping the simulation

*Step 1 – Create the object*

Now we can add Xsim and the auxiliary materials

- Simply add them to the git repository
Wrapping the simulation

Step 1 – Create the object

Now we can add XSim and the auxiliary materials

- Simply add them to the git repository

(a) Copy the code into the object folder:
   ```bash
cp -R ../XSim .
cp ../auxiliary_materials/* .
```

(b) Add and commit to the git repository:
   ```bash
git add *
git commit -m "Added the simulator and auxiliary materials"
```
Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance
Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance

But why?
Didn’t I just do that?
3 Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance

You only created the object!
Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance
Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance

And whenever you make changes

1 – Create object
2 – Import object
3 – Make changes
4 – Commit changes
3 Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance

You must commit them…
Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance

But don’t worry, we’ll cover these operations
Wrapping the simulation

Step 1 – Create the object

The object must now be imported

- Add the object to the local OCCAM instance

Run the command: `occam pull .`
Wrapping the simulation

Step 1 – Create the object

Verify that the object was successfully imported

- Go to the OCCAM web interface

On the search bar type the name of your simulator: e.g. XSim

Make sure that the simulator is found
Wrapping the simulation

Step 2 – Impl. build/run scripts

The next step is to make the object executable

1. Create the build and run scripts
2. Adding metadata to the object.json file
Wrapping the simulation

Step 2 – *Impl. build/run scripts*

Creating the build script

- **Write the build instructions**
  - similar to what we did before
  - needs to install some dependencies

(a) On the object directory open a file named build.sh

```
user@tutorial ~/exercises/ex2/simulator-XSim $ vim build.sh
```
Wrapping the simulation

Step 2 – Impl. build/run scripts

# Install compilation dependencies
apt-get install -y cmake libjsoncpp-dev
# Compile code
mkdir build
cd build
cmake ../XSim
make

(a) Check the file contents
Creating the launch script

- This file instructs OCCAM on how to run the simulation

(a) On the object directory open a file named launch.py
Wrapping the simulation

Step 2 – Impl. build/run scripts

This file tells OCCAM how to run the simulation

```python
import os
import json
# Helper file
from occam import Occam

# Get information about object
object = Occam.load()
# Gather paths
scripts_path = os.path.dirname(__file__)
job_path = os.getcwd()
object_path = "/occam/%s-%s" % (object.id(), object.revision())

OCCAM will place the simulator in this folder: /occam/<object id>-<object revision>
```

Look at the contents of the run script (ignore the commented code for now)

This is a helper script we created for you it parses the OCCAM run information
③ Wrapping the simulation

Step 2 – Impl. build/run scripts

# Set variables
runner_script = os.path.join(object_path, "runner.py")
build_folder = os.path.join(object_path, "build")
input_file = os.path.join(object_path, "default_program.m")
simulation = os.path.join(object_path, "simulation.py")
simulation_configuration = os.path.join(object_path, "default_simulation.json")

output_file = "statistics.json"

These are the inputs for the run command (remember when you executed the simulation before)

(continues on the next slide)
# Setup run command
run_sim_command = [
    "python ", runner_script,
    " --simulation ", simulation,
    " --build-folder ", build_folder,
    " --input-program ", input_file,
    " --simulation-config ", simulation_configuration,
    " --output-file ", output_file
]
command= ' '.join(run_sim_command)

# Pass run command to OCCAM
Occam.report(command)
Wrapping the simulation

Step 2 – Impl. build/run scripts

Still missing the metadata about these scripts:

- OCCAM checks the object.json file for it
  1. What is the environment where it runs
  2. Which is the build script
  3. Which is the run script

The environment is a Docker image where the simulator will be build and run
Wrapping the simulation

Step 2 – Impl. build/run scripts

(a) Open the object.json file
3. Wrapping the simulation

Step 2 – Impl. build/run scripts

Set the environment

- OCCAM needs to know:
  - the environment
  - the architecture

```json
{
  ...,  
  "architecture":"x86-64", 
  "environment":"ubuntu:SSTOS"
}
```

An environment that runs in x86-64 and provides ubuntu:SSTOS has already been provided. It contains SST ready to be used.
Wrapping the simulation

Step 2 – Impl. build/run scripts

- In order to build and run the simulator, OCCAM needs to know:
  - the environment
  - the architecture

(a) Add the following to the metadata in object.json

```json
{
  ..., 
  "architecture":"x86-64",
  "environment":"ubuntu:SSTOS",
}
```
3 Wrapping the simulation

Step 2 – Impl. build/run scripts

You can find the environment on the OCCAM web interface

(a) On the search bar type the name of the environment: SSTOS

(b) Click search
Wrapping the simulation

Step 2 – Impl. build/run scripts

(a) Add the build and run script locations

```json
{

  ...,

  "architecture": "x86-64",
  "environment": "ubuntu:SSTOS",
  "build": {
    "command": "sh build.sh"
  },

  "run": {
    "script": "launch.py",
    "version": "3.3",
    "language": "python"
  }

}```
Wrapping the simulation

Step 2 – Impl. build/run scripts

```json
{
...
  "architecture": "x86-64",
  "environment": "ubuntu:SSTOS",
  "build": {
    "command": "sh build.sh"
  },
  "run": {
    "script": "launch.py",
    "version": "3.3",
    "language": "python"
  }
}
```
Wrapping the simulation

Step 2 – Impl. build/run scripts

```json
{
  ...,
  "architecture": "x86-64",
  "environment": "ubuntu:SSTOS",
  "build": {
    "command": "sh build.sh"
  },
  "run": {
    "script": "launch.py",
    "version": "3.3",
    "language": "python"
  }
}
```

More complex (and detailed) form that specifies the version and language of the run script.
③ Wrapping the simulation

Step 2 – Impl. build/run scripts

(a) Validate the json file you modified by running the command:
cat object.json | python –m json.tool
Wrapping the simulation

Step 2 – Impl. build/run scripts

json.tool is a useful python tool that will let you know if you made a syntax mistake in your json files.

```bash
user@tutorial ~/exercises/ex2/simulator-XSim $ cat object.json | python -m json.tool
{
    "architecture": "x86-64",
    "build": {
        "command": "sh build.sh"
    },
    "environment": "ubuntu:SSTOS",
    "id": "9fdee3ac-4abd-11e7-88cf-080027de086d",
    "name": "XSim",
    "run": {
        "language": "python",
        "script": "launch.py",
        "version": "3.3"
    },
    "type": "simulator"
}
```
Wrapping the simulation

Step 2 – Impl. build/run scripts

With all required modification made

- The OCCAM object can be executed
- But first, changes must be committed

(a) Add the changes to git:
git add object.json

(b) Commit the changes to git:
git commit –m “Added the build and run scripts”
3 Wrapping the simulation

Step 2 – Impl. build/run scripts

Now commit into OCCAM

- Check that it builds correctly!

(a) Commit changes in the object into OCCAM

(b) Build the object
Wrapping the simulation

Step 2 – Impl. build/run scripts

Now commit into OCCAM

- Check that it builds correctly!

(a) Check if the object built correctly
3 Wrapping the simulation

Step 2 – Impl. build/run scripts

Check your new OCCAM object

(a) Click your username to go back to your worksets

(b) Create a new workset
Wrapping the simulation

Step 2 – Impl. build/run scripts

(a) In the bottom of the workset page create a new experiment
Wrapping the simulation

Step 2 – Impl. build/run scripts

Check your new OCCAM object

(a) Click the item in the empty workflow
(b) Add your simulator
(c) Click attach
Wrapping the simulation

Step 2 – Impl. build/run scripts

Check your new OCCAM object

Here is your creation, however, it has neither input nor output. Yet!
The next step is to add input/output to the object
1. Create the output schema
2. Add the metadata for both input and output
3. Update the run script to use these interfaces
③ Wrapping the simulation

Step 3 – Add input/output

First let’s create the output schema

- They describe the data output by the object
  - Allows OCCAM to interpret the output

- Is implemented according to the simulation
  - Follows OCCAM specification for schemas
Wrapping the simulation

Step 3 – Add input/output

Output schemas are very simple JSON files, e.g.

This simulator output:

```
{
  "name":"Experiment 1",
  "results": [
    {"value":5},
    {"value":10},
    {"value":15}
  ],
  "stats": {
    "mean_value":10
  }
}
```

Is represented by this schema:

```
{
  "name": {
    "type": "string"
  },
  "results": [
    {"value": {"type": "int"}}
  ],
  "stats": {
    "mean_value": {"type": "int"}
  }
}
```
Wrapping the simulation

Step 3 – Add input/output

Open the output schema file

(a) On the simulator directory open the file named output_schema.json

user@tutorial ~/exercises/ex2/simulator-XSim $ vim output_schema.json
Wrapping the simulation

Step 3 – Add input/output

The output schema file

```json
{
  "name": {"type": "string"},
  "experiments": [{
    "clock_frequency": {"type": "string"},
  }, ...
  (continues)
}
```

- The name of the output
- An array of several experiments
- The clock frequency

(a) Copy the output schema to the file you just created
The output schema file

... (continued)

"registers": [{
  "r0": {"type": "int"},
  "r1": {"type": "int"},
  "r2": {"type": "int"},
  "r3": {"type": "int"},
  "r4": {"type": "int"},
  "r5": {"type": "int"},
  "r6": {"type": "int"},
  "r7": {"type": "int"}
},
... (continues)
Wrapping the simulation

Step 3 – Add input/output

The output schema file

... (continued)

"stats": [{
  "add": {"type": "int"},
  "sub": {"type": "int"},
  "and": {"type": "int"},
  "nor": {"type": "int"},
  "div": {"type": "int"},
  "mul": {"type": "int"},
  "mod": {"type": "int"},
  "exp": {"type": "int"},
... (continues)

- Number of calls to each instruction
- Total number of instructions called
- Number of CPU cycles executed
Wrapping the simulation

Step 3 – Add input/output

The output schema file

... (continued)

"lw":{"type": "int"},
"sw":{"type": "int"},
"liz":{"type": "int"},
"lis":{"type": "int"},
"lui":{"type": "int"},
"bp":{"type": "int"},
"bn":{"type": "int"},
"bx":{"type": "int"},
"bz":{"type": "int"},

... (continues)
Wrapping the simulation

Step 3 – Add input/output

The output schema file

... (continued)

"jr":{"type": "int"},
"jal":{"type": "int"},
"j":{"type": "int"},
"halt":{"type": "int"},
"put":{"type": "int"},
"instructions":{"type": "int"},
"cycles":{"type": "int"}
Wrapping the simulation

Step 3 – Add input/output

Now add the metadata to the object specification

(a) Open the object.json file

```bash
user@tutorial ~/exercises/ex2/simulator-XSim $ vim object.json
```
First add the input section to the metadata

- This section describes the input information

(a) Add the following input to the `object.json` file

```
...,  
  "inputs": [  
    {  
      "type": "program/XSim"  
    }  
  ]  
...
```

Types are user defined, i.e. there are no predefined types

Make sure that this comma is there!
Wrapping the simulation

Step 3 – Add input/output

Then add the input section to the metadata

- This section describes the output information

Add the following input to the `object.json` file

```json
...
  "outputs": [
    {
      "file": "statistics.json",
      "createIn": "output_dir",
      "type": "application/json",
      "schema": "output_schema.json"
    }
  ]
...
```
Wrapping the simulation

Step 3 – Add input/output

Adding an output section to the object.json file

- This section contains the output information

```json
...,
  "outputs": [
    {
      "file": "statistics.json",
      "createIn": "output_dir",
      "type": "application/json",
      "schema": "output_schema.json"
    }
  ]
...
```

Name of output file
Wrapping the simulation

Step 3 – Add input/output

Adding an output section to the object.json file

- This section contains the output information

```json
...
"outputs": [
{
  "file": "statistics.json",
  "createIn": "output_dir",
  "type": "application/json",
  "schema": ""
}
]
...
```

Name of the directory where the output will be written
Wrapping the simulation

Step 3 – Add input/output

Adding an output section to the object.json file

- This section contains the output information

```json
...,  
  "outputs": [  
    {  
      "file": "statistics.json",  
      "createIn": "output_dir",  
      "type": "application/json",  
      "schema": "output_schema.json"  
    }  
  ]  
...  
```

Type of the output object
Wrapping the simulation

Step 3 – Add input/output

Adding an output section to the object.json file

- This section contains the output information

```json
...
  "outputs": [
    {
      "file": "statistics.json",
      "createIn": "output_dir",
      "type": "application/json",
      "schema": "output_schema.json"
    }
  ]
...
```

The schema you just created
Now the object has an input and an output

- But the run script
  - Is running using the default input
    \[
    \text{input\_file} = \text{os.path.join(object\_path, "default\_program.m")}
    \]
  - Is writing the output in the wrong place
    \[
    \text{output\_file} = "\text{statistics.json}"
    \]

We need to modify the run script

```bash
user@tutorial ~/exercises/ex2/simulator-XSim $ vim launch.py
```

(a) Open the launch.py script
Step 3 – Add input/output

input_file = os.path.join(object_path, "default_program.m")
# Get input from OCCAM
inputs = object.inputs("program/XSim")
if len(inputs) > 0:
    files = inputs[0].files()
    if len(files) > 0:
        input_file = files[0]

(a) Find the input file declaration in the run script
(b) Remove the comments on the following code

This will check if an input file was given to the simulator
Step 3 – Add input/output

(a) Find the output file declaration in the run script

```python
# Output file
output_file="statistics.json"

# Get output path from OCCAM (inputs <ty>,<output directory>, <output filename>)
output_file = object.output_path("application/json","output_dir", "statistics.json")
```

(b) Remove the comments on the following code

This will get the path where OCCAM is expecting the output file
Wrapping the simulation

Step 3 — Add input/output

(a) Validate the json file you modified by running the command:
cat object.json | python -m json.tool
3 Wrapping the simulation

Step 3 – Add input/output

The object has input and output

- But still needs to be committed

(a) Add the files you created/modified to the git repository:
git add object.json launch.py

(b) Commit the git repository:
git commit –am “Implemented the input and output interfaces”
3 Wrapping the simulation

Step 3 – Add input/output

The object has input and output

- But still needs to be committed

(a) Commit changes in the object into OCCAM

```
user@tutorial ~/exercises/ex2/simulator-XSim $ occam commit
Committing object
* Committing resources
* Updating database record.
* No changes.
* Current revision: 4a02f0764ed8e669660f5941fb52423851cc3aa0
* Done: Updated object 9fdee3ac-4abd-11e7-88cf-080027de086d
```

(b) Build the object

```
user@tutorial ~/exercises/ex2/simulator-XSim $ occam build
```
3 Wrapping the simulation

Step 3 – Add input/output

Try running the new version of the OCCAM object
Wrapping the simulation

Step 3 – Add input/output

(a) Go back to the Exercise 2 workset

(b) In the bottom of the workset page create a new experiment
③ Wrapping the simulation

Step 3 – Add input/output

Add your simulator to a workflow

(a) Click the item in the empty workflow

(b) Add your simulator

(c) Click attach
3 Wrapping the simulation

Step 3 – Add input/output

Add input to your simulation

Now the simulator has input and output
3 Wrapping the simulation

Step 3 – Add input/output

Add input to your simulation

But for now, you’re still missing the configuration options
③ Wrapping the simulation

Step 3 – Add input/output

Add input to your simulation

(a) Click the plus sign to add an input
Wrapping the simulation

Step 3 – Add input/output

Add input to your simulation

(a) Type program/XSim on the top box

(b) In the middle box find the XSim-Program

(c) Click attach
3 Wrapping the simulation

Step 3 – Add input/output

Add input to your simulation

Now you can run the simulation!
Wrapping the simulation

Step 4 – Add configurations

Finally add configuration schemas
Finally add configuration schemas

- Simulations often have configurable parameters
  - Allow to modify the simulated models

- Software creators know available configurations
  - Configuration schemas expose them
  - End-users use the web interface to change them

- Configurations are part of the object
  - They are described in a file using JSON
Wrapping the simulation

Step 4 – Add configurations

Configuration schema structure:

```json
{
  <group name>:{
    "label":<label>,
    <option name>:{
      "type": <type>,
      "default": <default>,
      "label":<label>,
      "description":<description>
    }
  }
}
```

Groups help organizing options and must be valid variable names (no spaces special characters)
Configuration schema structure:

```json
{
  <group name>:{
    "label":<label>,
    <option name>:{
      "type": <type>,
      "default": <default>,
      "label":<label>,
      "description":<description>
    }
  }
}
```

This is a configurable option and must be a valid variable name (no spaces special characters)
Wrapping the simulation

Step 4 – Add configurations

Configuration schema structure:

```json
{
    <group name>:{
        "label":<label>,
        <option name>:{
            "type": <type>,
            "default": <default>,
            "label":<label>,
            "description":<description>
        }
    }
}
```
Configuration schema structure:

```json
{
    <group name>: {
        "label": <label>,
        <option name>: {
            "type": <type>,
            "default": <default>,
            "label": <label>,
            "description": <description>
        }
    }
}
```

The type is used for type-checking when assembling experiments.
Wrapping the simulation

Step 4 – Add configurations

Configuration schema structure:

```json
{
    <group name>:{
        "label":<label>,
        <option name>:{
            "type": <type>,
            "default": <default>,
            "label":<label>,
            "description":<description>
        }
    }
}
```

The default variable value 107
Configuration schema structure:

```json
{
    <group name>):{
        "label":<label>,
        <option name>:{
            "type": <type>,
            "default": <default>,
            "label":<label>,
            "description":<description>
        }
    }
}
```
Wrapping the simulation

Step 4 – Add configurations

Configuration and output schemas are similar

- The main differences are
  - Configurations are rendered as forms
    - By the Web Interface
  - The forms used to generate the actual configurations
    - Accessible by the simulator on runtime
Wrapping the simulation

Step 4 – Add configurations

Configuration generated by OCCAM

This schema

Generates this configuration (using the default values)

```json
{
    "cpu": {
        "label": "XSim",
        "clock_frequency": {
            "type": "string",
            "default": "2MHz",
            "label": "The frequency of the CPU"
        }
    }
}
```

```json
{
    "cpu": {
        "clock_frequency": "2MHz"
    }
}
```
Wrapping the simulation

Step 4 – Add configurations

Example of the rendering:

```json
{  
    "cpu": {  
        "label": "XSim",  
        "clock_frequency": {  
            "type": "string",  
            "default": "2MHz",  
            "label": "The frequency of the CPU"  
        }  
    }  
}
```
Wrapping the simulation

Creating the schema for the simulator

Step 4 – Add configurations

Open the file named “configuration_schema.json”
Wrapping the simulation

Step 4 – Add configurations

```json
{
    "name": {
        "type": "string",
        "label": "Name of experiment",
        "default": "Xsim-results"
    },
    "cpu": {
        "label": "XSim",
        "clock_frequency": {
            "label": "The frequency of the CPU",
            "type": "string",
            "default": "2MHz"
        }
    }
}
... (continues)
```
Wrapping the simulation

**Step 4 – Add configurations**

... (continued)

```json
"memory": {
    "label": "Memory options",

    "backend.mem_size": {
        "label": "Memory Size",
        "type": "string",
        "default": "2MiB"
    }
},

... (continues)
```
3 Wrapping the simulation

Step 4 – Add configurations

... (continued)

"clock":{
   "label":"Clock frequency (Can be a comma separated list of frequencies)",
   "type":"string",
   "default":"200Hz"
},

"backend":{
   "label":"Backend",
   "type": ["memHierarchy.simpleMem"],
   "default": "memHierarchy.simpleMem"
},

... (continues)
Wrapping the simulation

Step 4 – Add configurations

... (continued)

"simpleMem": {
  "label": "SimpleMem",
  "verbose": {
    "type": "int",
    "default": 0,
    "label": "Verbose"
  },

  "access_time": {
    "type": "string",
    "default": "100ns",
    "label": "Access time"
  }
}

Memory options

- Memory Size: 2MiB
- Clock frequency: 200Hz
- Backend: 
  - MemHierarchy.simpleMem

SimpleMem

- Verbose: 0
- Access time: 100ns
③ Wrapping the simulation

Step 4 – Add configurations

Now add the metadata to the object specification

(a) Open the object.json file

```
user@tutorial ~/exercises/ex2/simulator-XSim $ vim object.json
```
Let OCCAM know about the configurations schema

Add the configurations to the object metadata (object.json file)

```json
...
"configurations":[
{
  "schema": "simulation_schema.json",
  "file": "simulation.json",
  "type": "application/json",
  "name": "Simulation configuration"
}
]
...
```
Wrapping the simulation

Step 4 – Add configurations

Let OCCAM know about the configuration schema

```json
...
"configurations": [
  {
    "schema": "simulation_schema.json",
    "file": "simulation.json",
    "type": "application/json",
    "name": "Simulation configuration"
  }
]
...
```

The name of the file containing the schema

File created with the configuration
3 Wrapping the simulation

Step 4 – Add configurations

Let OCCAM know about the configuration schema

```json
... "configurations": [
  {
    "schema": "simulation_schema.json",
    "file": "simulation.json",
    "type": "application/json",
    "name": "Simulation configuration"
  }
]
..."
Finally we need to change the run script

- Similarly to the input/output
  - We will get the configuration options from OCCAM

(a) Open the launch.py script
Wrapping the simulation

Step 4 – Add configurations

# Default configuration parameters
simulation_configuration = os.path.join(object_path, 
"default_simulation.json")

# Get configuration parameters from OCCAM
simulation_configuration = 
object.configuration_file("Simulation configuration")

(a) Find the configuration file declaration in the run script

(b) Uncomment the code

This will get the configuration file path from OCCAM
Wrapping the simulation

Step 4 – Add configurations

Validate the json file you modified by running the command:
cat object.json | python -m json.tool
3 Wrap the simulation

**Step 4 – Add configurations**

Add the configuration schema to the object

- Commit to git and OCCAM

(a) Add files to the git repository:
   `git add object.json launch.py`

```
user@tutorial ~/exercises/ex2/simulator-XSim $ git add object.json launch.py
```

(b) Commit the changes to git:
   `git commit -m "Added a configuration schema"`

```
user@tutorial ~/exercises/ex2/simulator-XSim $ git commit -am "Added the configuration schema"
```

(c) Commit the changes to OCCAM:
   `occam commit`

```
user@tutorial ~/exercises/ex2/simulator-XSim $ occam commit
Committing object
  * Committing resources
```
Wrapping the simulation

Step 4 – Add configurations

Commit changes OCCAM

- Update the object within the system

(a) Commit changes in the object into OCCAM

(b) Build the object
Wrapping the simulation

Step 4 – Add configurations

On your own

- Create an experiment using the XSim
- Check the available configuration options
  - Compare them with the schema you created
In this exercise

- You took the role of a developer
  - Integrating his SST simulator in OCCAM
- You learned the necessary steps to wrap it
  - Creating all the scripts and schemas required
Acknowledgments

We gratefully acknowledge the support and collaboration of our research partners.

Portions of this material is based in part upon work supported by the National Science Foundation under Grant Numbers CCF-142331, CNS-1012070, ACI-1535232, CNS-1305220 and CCF-1148646. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.